Effective and Reliable Microservices

Oleg Anastasyev
oa@ok.ru
@m0nstermind

Top 10 countries by OK audience

43 mln
Russia

2,8 mln
Belarus

2,6 mln
Kazakhstan

2,1 mln
Uzbekistan

1,7 mln
Germany

1,5 mln
Moldova

1,1 mln
Armenia

760 000
Tajikistan

760 000
Georgia

Source: OK internal data, MAU, February 2020

OK.ru

9k

machines

7

clouds

48k

tasks

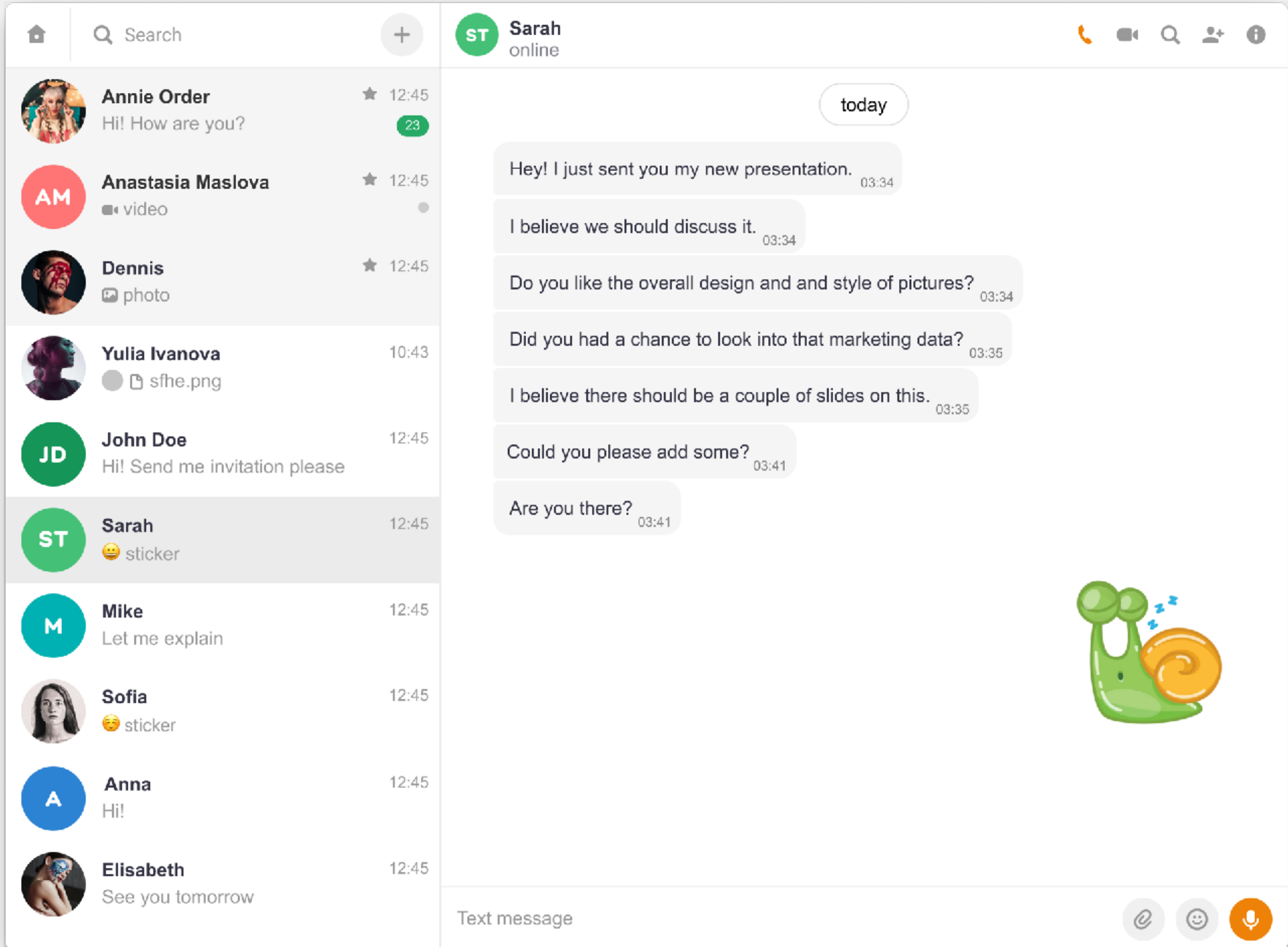
300

services

Messenger as an example

5% chats make **95%** requests

80% freshest 13 messages



Chat Messages: storage

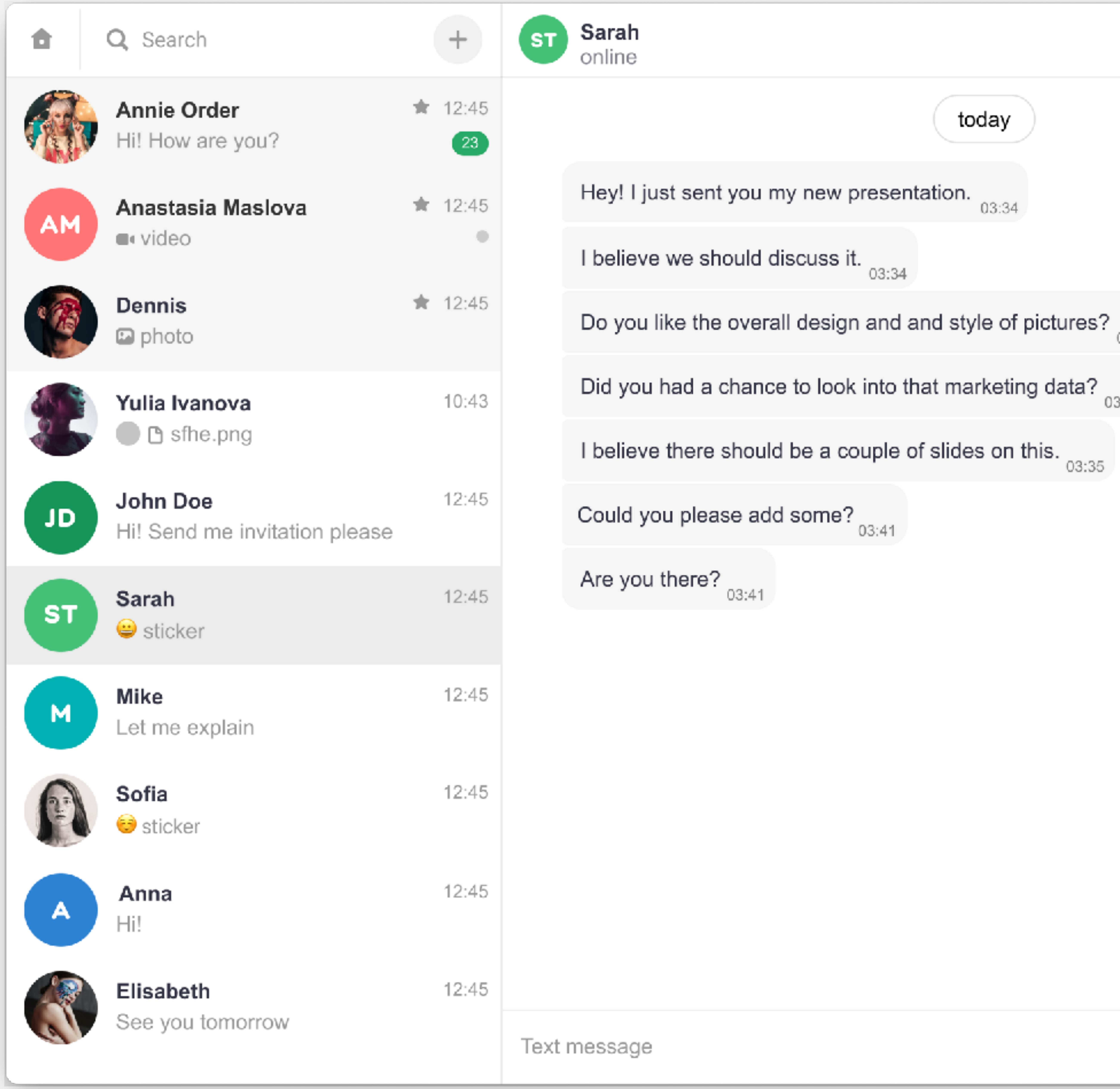
600 bi
messages

5 bi
chats

100 TB
data

100+ Kops
reads

10+ Kops
writes



Chat Messages: full text search

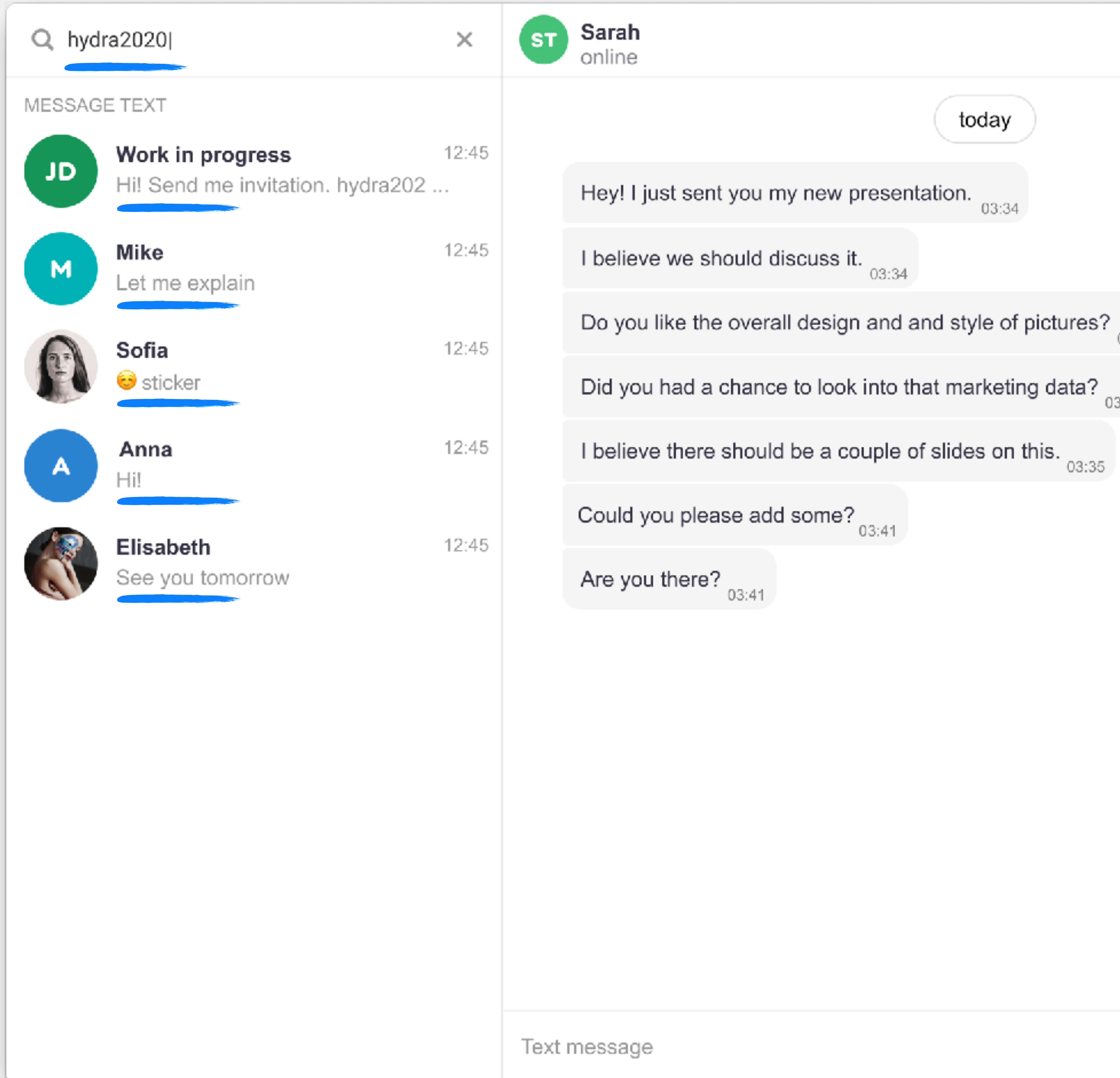
600 bi messages

5 bi chats

100 TB data

100+ Kops reads

10+ Kops writes



Chat Messages Storage Service

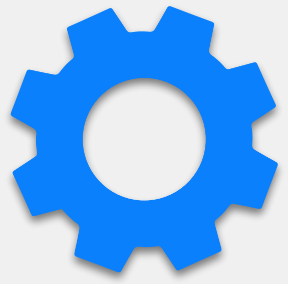
operations:

- getMessages(viewer, chat, from, to)
- getLastMessages(viewer, chats)
- add(chat, message)
- search(viewer, text)
- indexMessages()

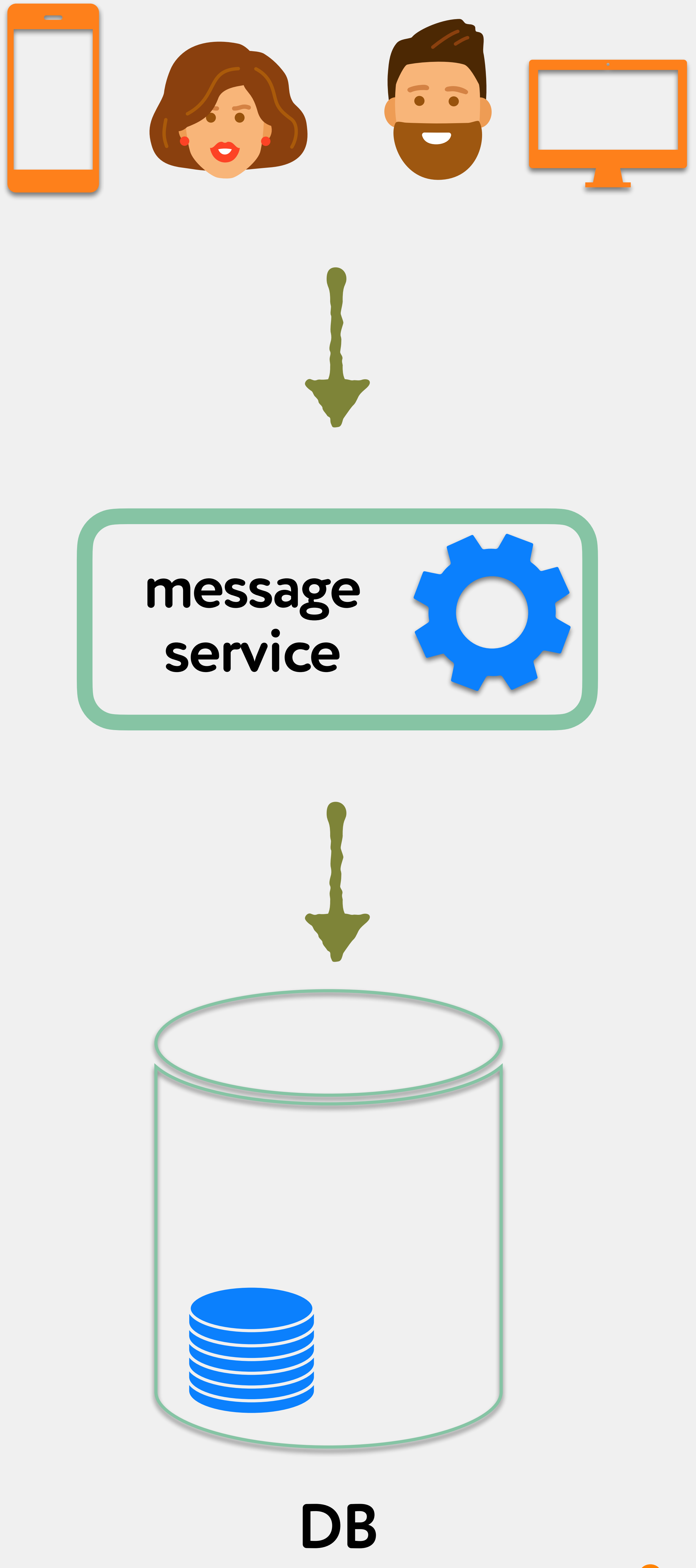
	ID	From	Type	Text	Attach[]
1	14:30	Vader	TXT	Hello Luke, calling	
1	14:31	Vader	VIDC	callto: Luke, miss	
1	14:32	Procter	SPAM	someth	some.gif
1	14:35	Luke		Who is this ?	

```
CREATE TABLE Messages (
    chatId, msgId

    user, type, text, attachments[], terminal, deletedBy[], replyTo,...

    PRIMARY KEY ( chatId, msgId )
)
```

Microservice Architecture

 Application Logic

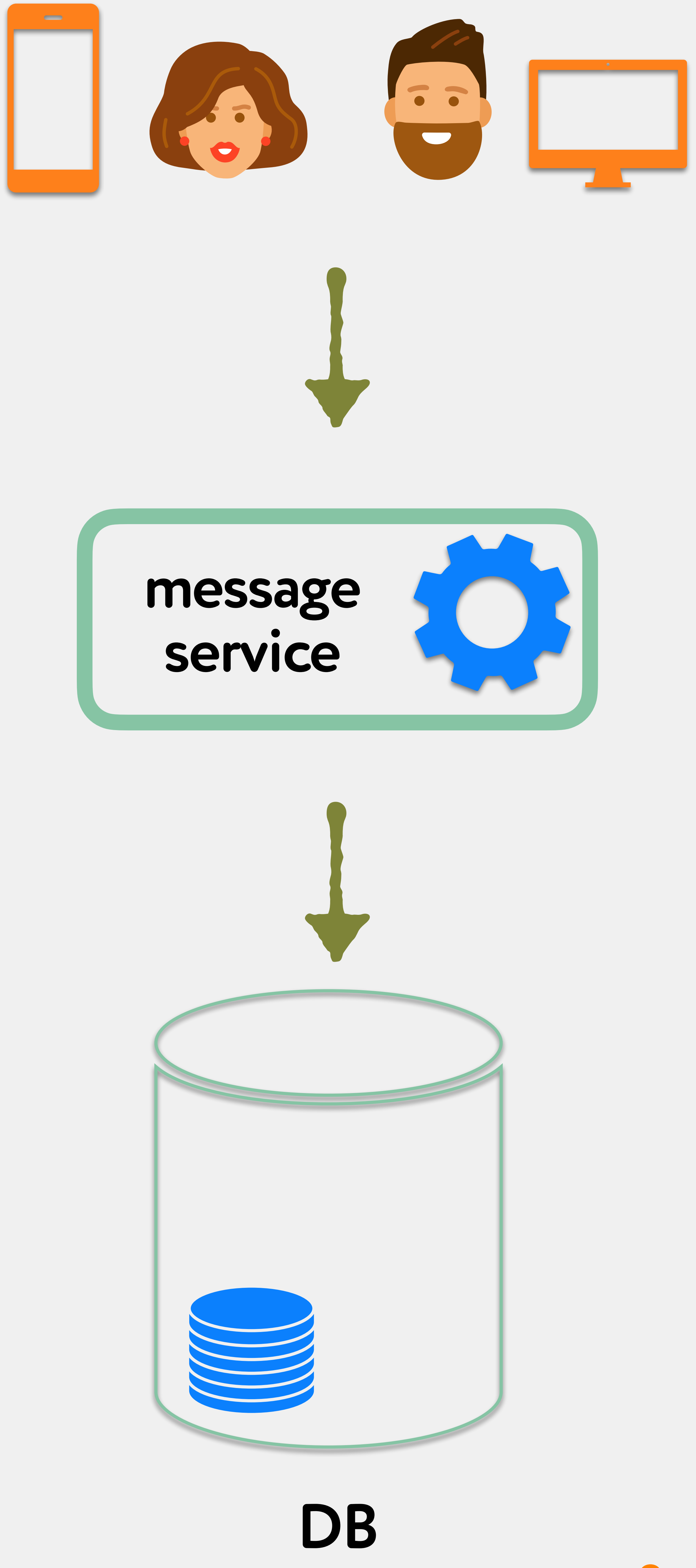
 Application State (data)



Microservice Architecture

- getMessages(viewer, chat, from, to)

```
SELECT FROM Messages
    WHERE chatId = ? AND
    msgId BETWEEN :from AND :to
```



Microservice Architecture

- getMessages(viewer, chat, from, to)

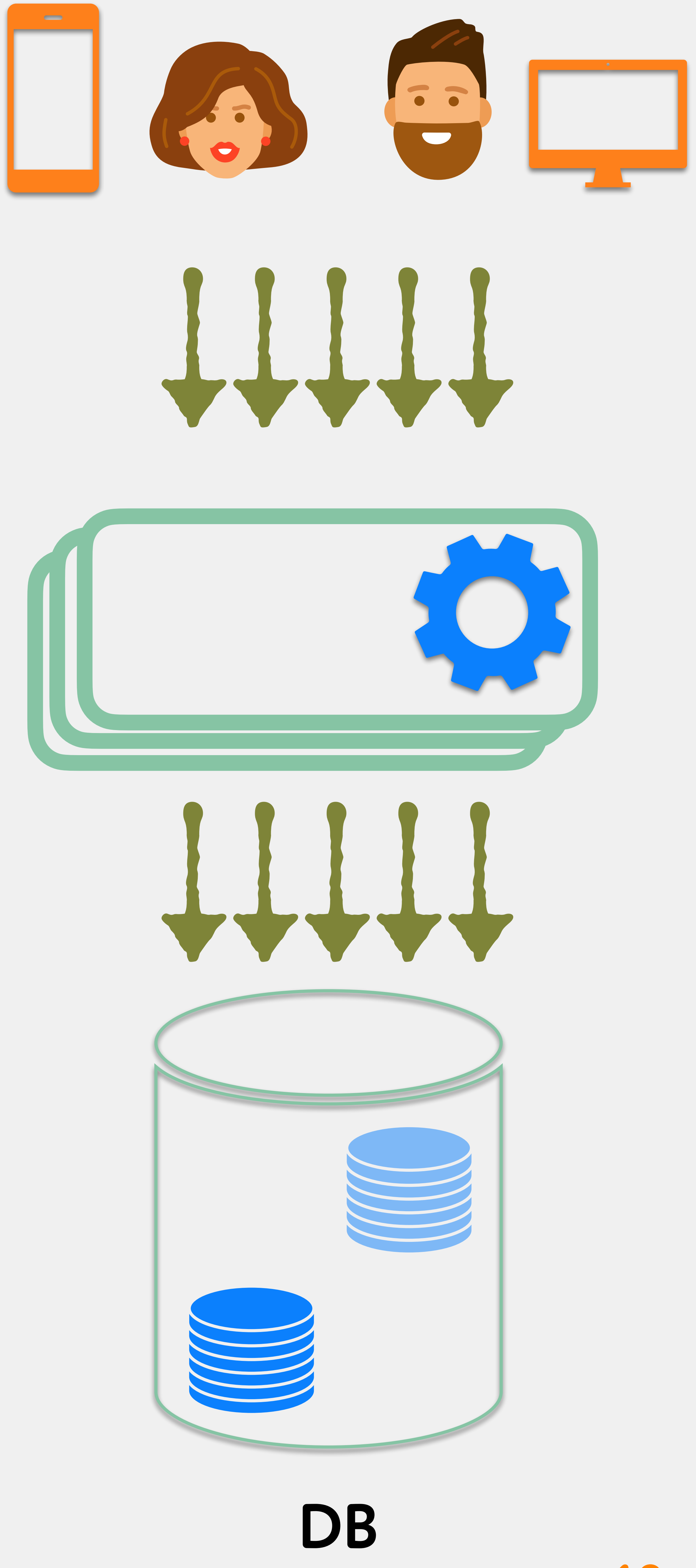
100+ k/sec

5% chats make **95%** requests

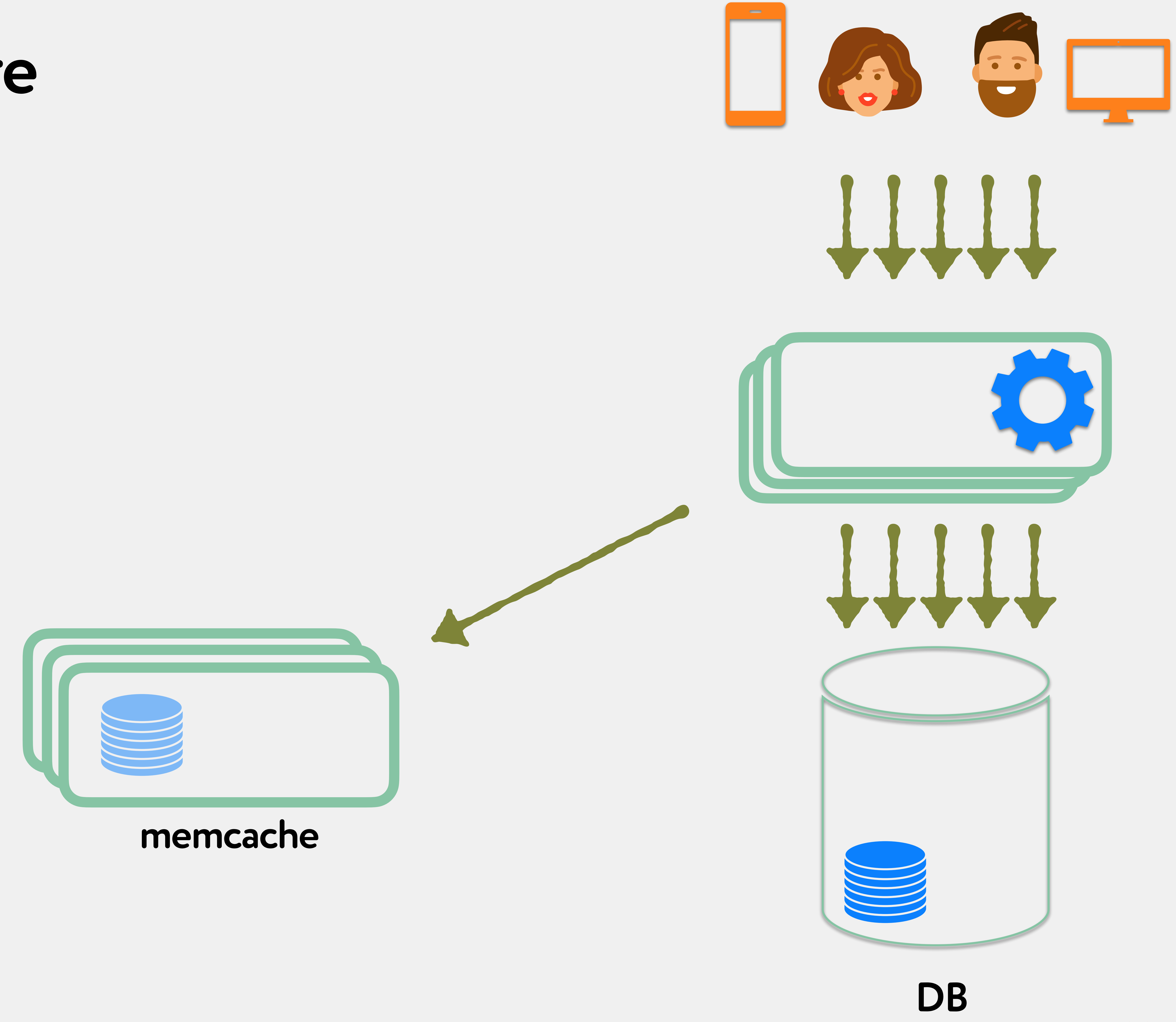
- getLastMessages(viewer, chats)
- indexMessages()

100% < 1%

chats requests



Microservice Architecture

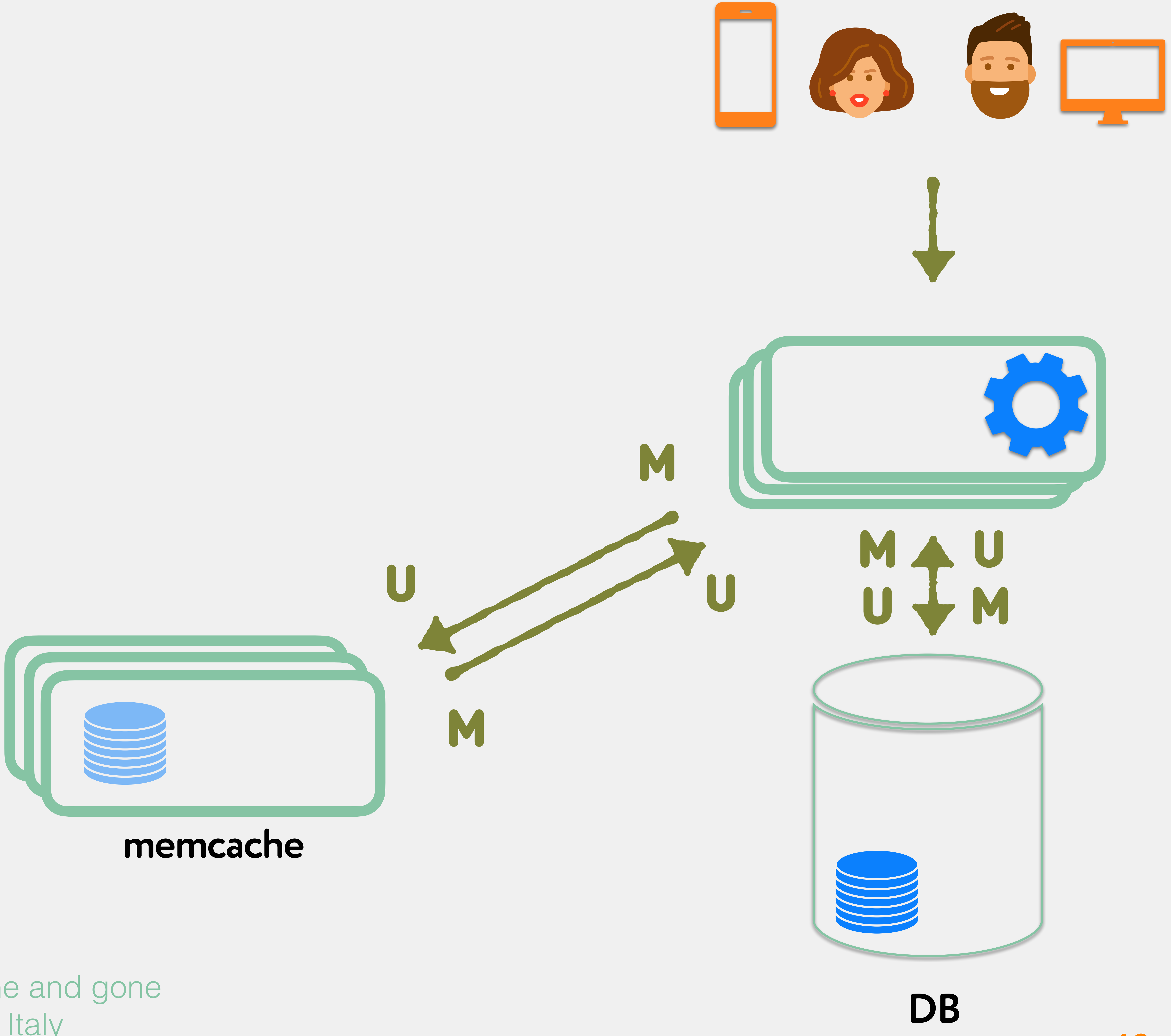


Microservices: costs

- CPU: (Un) Marshalling

+85%
CPU load[1]

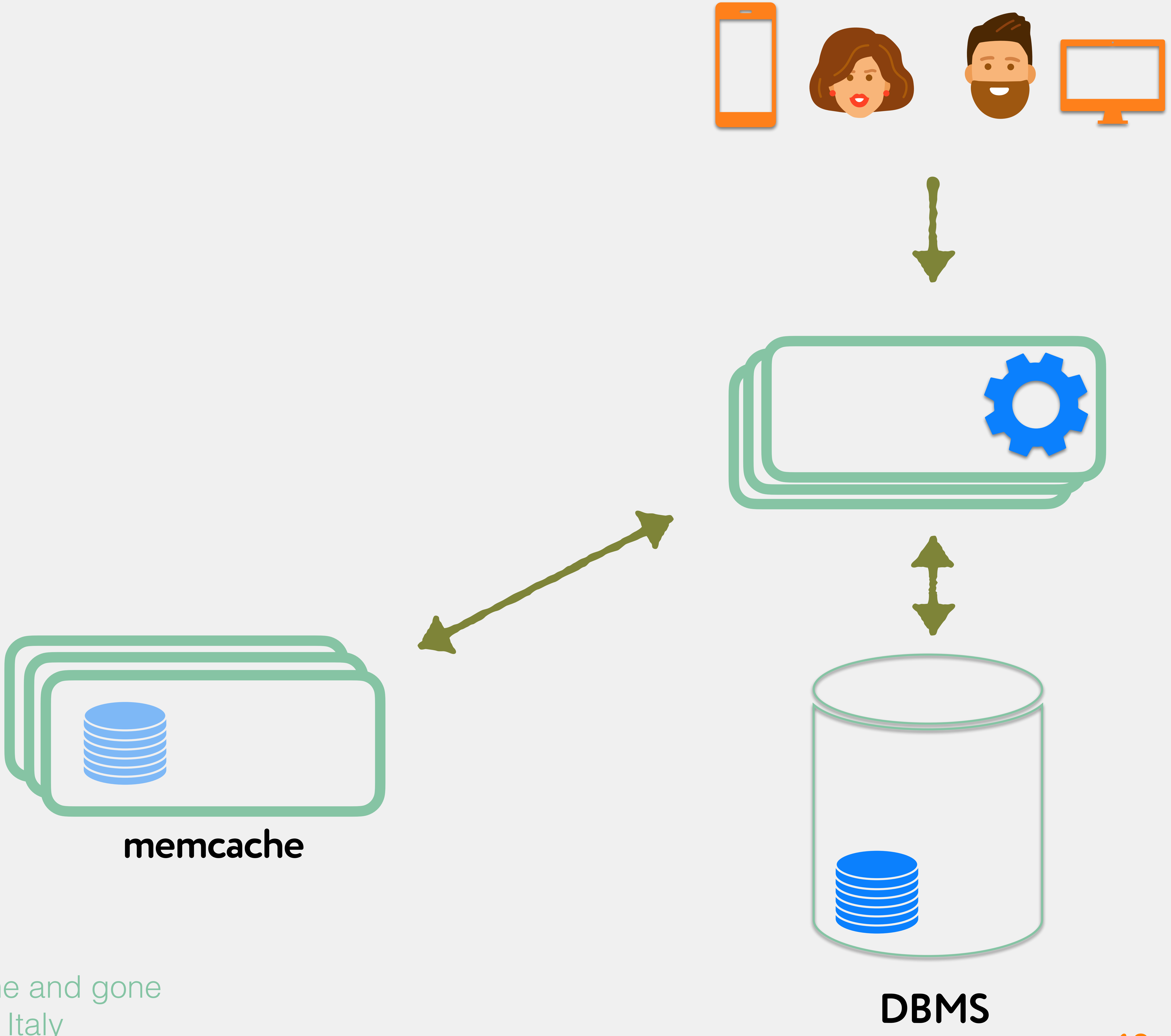
+27%
median latency[1]



(1) Fast key-value stores: An idea whose time has come and gone
Adya et al. HotOS '19, May 13–15, 2019, Bertinoro, Italy

Microservices: costs

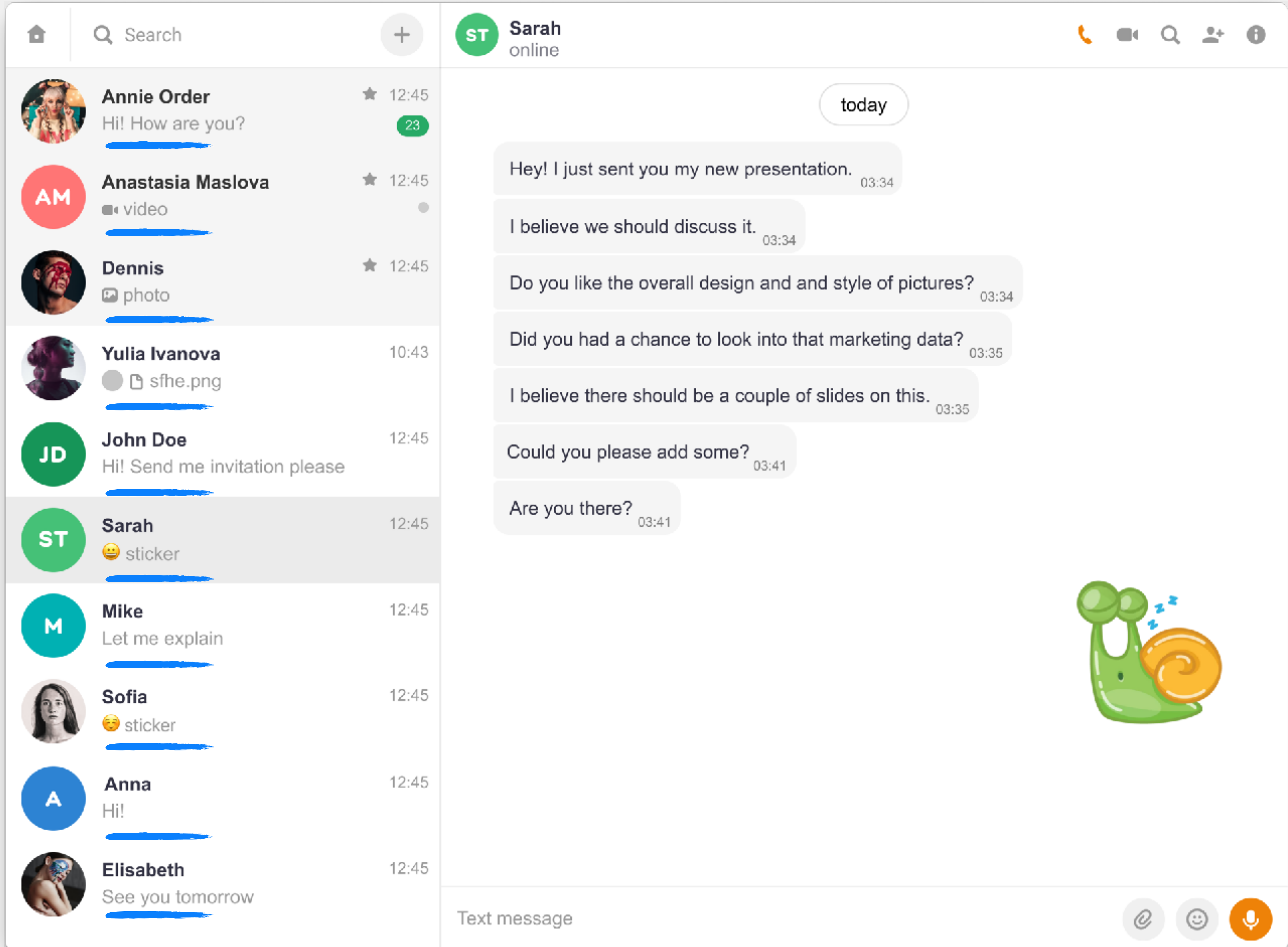
- CPU: (Un) Marshalling
- Overreads, overwrites



(1) Fast key-value stores: An idea whose time has come and gone
Adya et al. HotOS '19, May 13–15, 2019, Bertinoro, Italy

Microservices: costs

- CPU: (Un) Marshalling
- Overreads, overwrites



Microservices: costs

- **CPU: (Un) Marshalling**
- **Overreads, overwrites**

if we use only

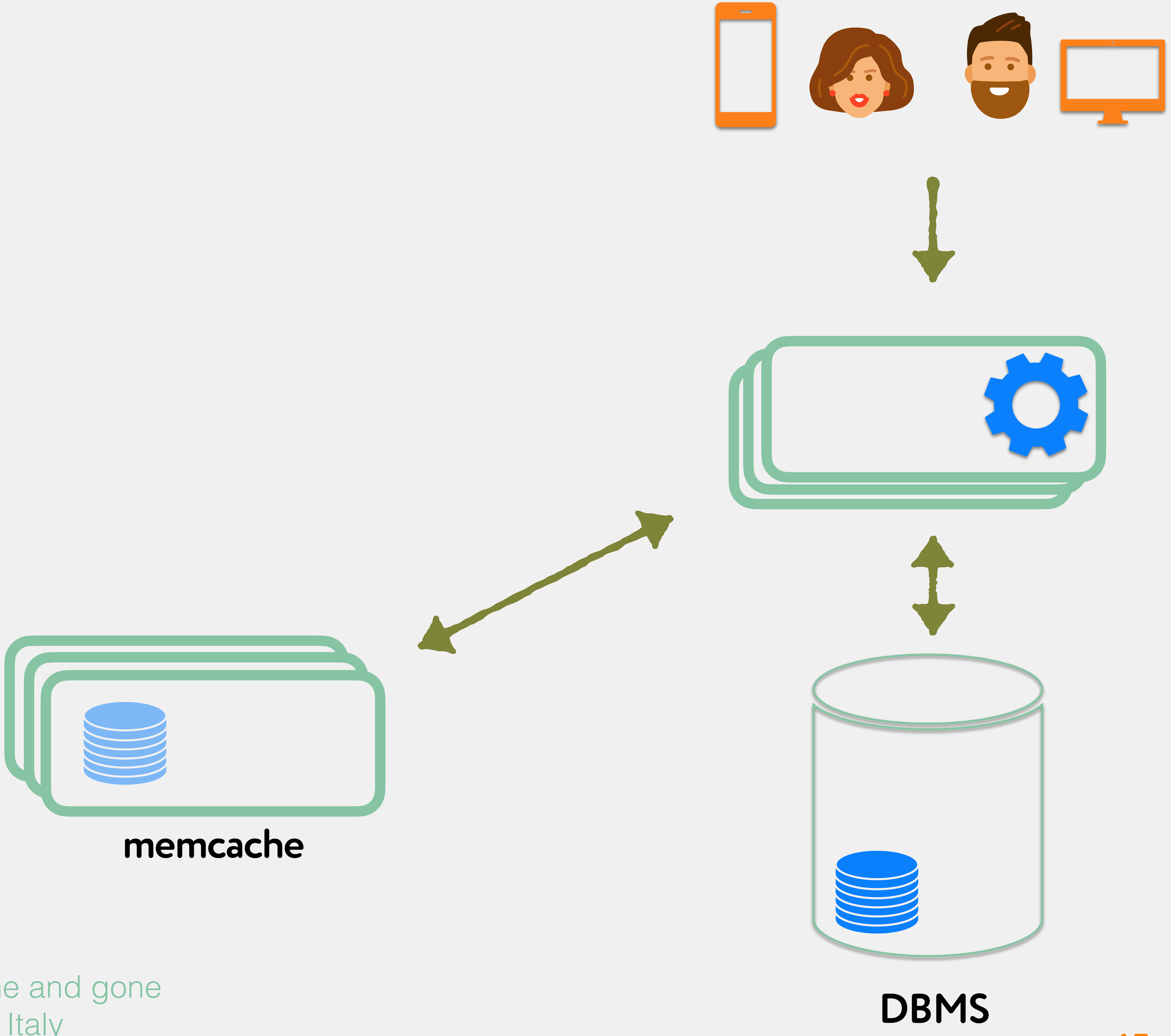
10%

of data read, then up to

+46% CPU

+86% Net

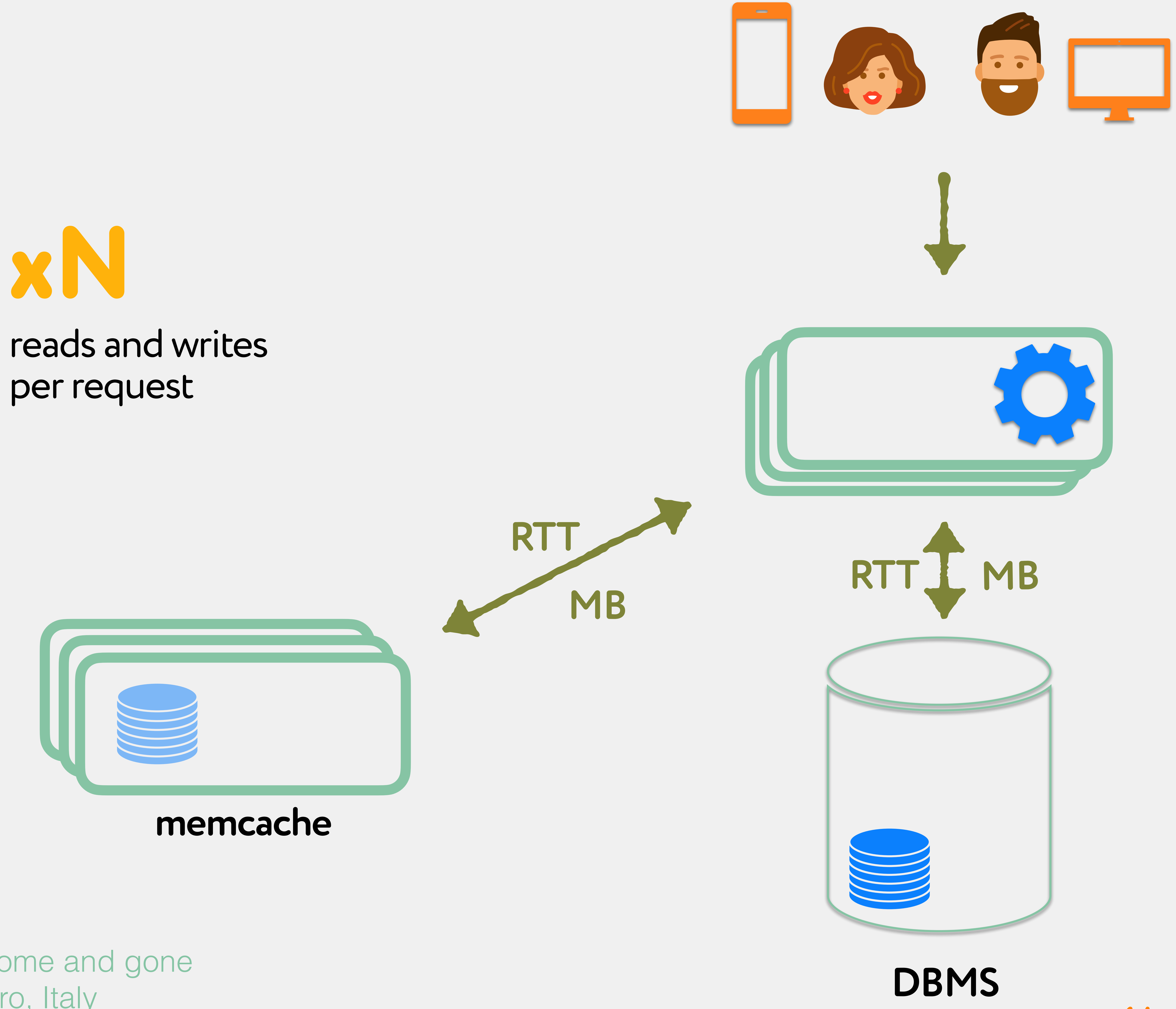
are wasted [1)]



(1) Fast key-value stores: An idea whose time has come and gone
Adya et al. HotOS '19, May 13–15, 2019, Bertinoro, Italy

Microservices: costs

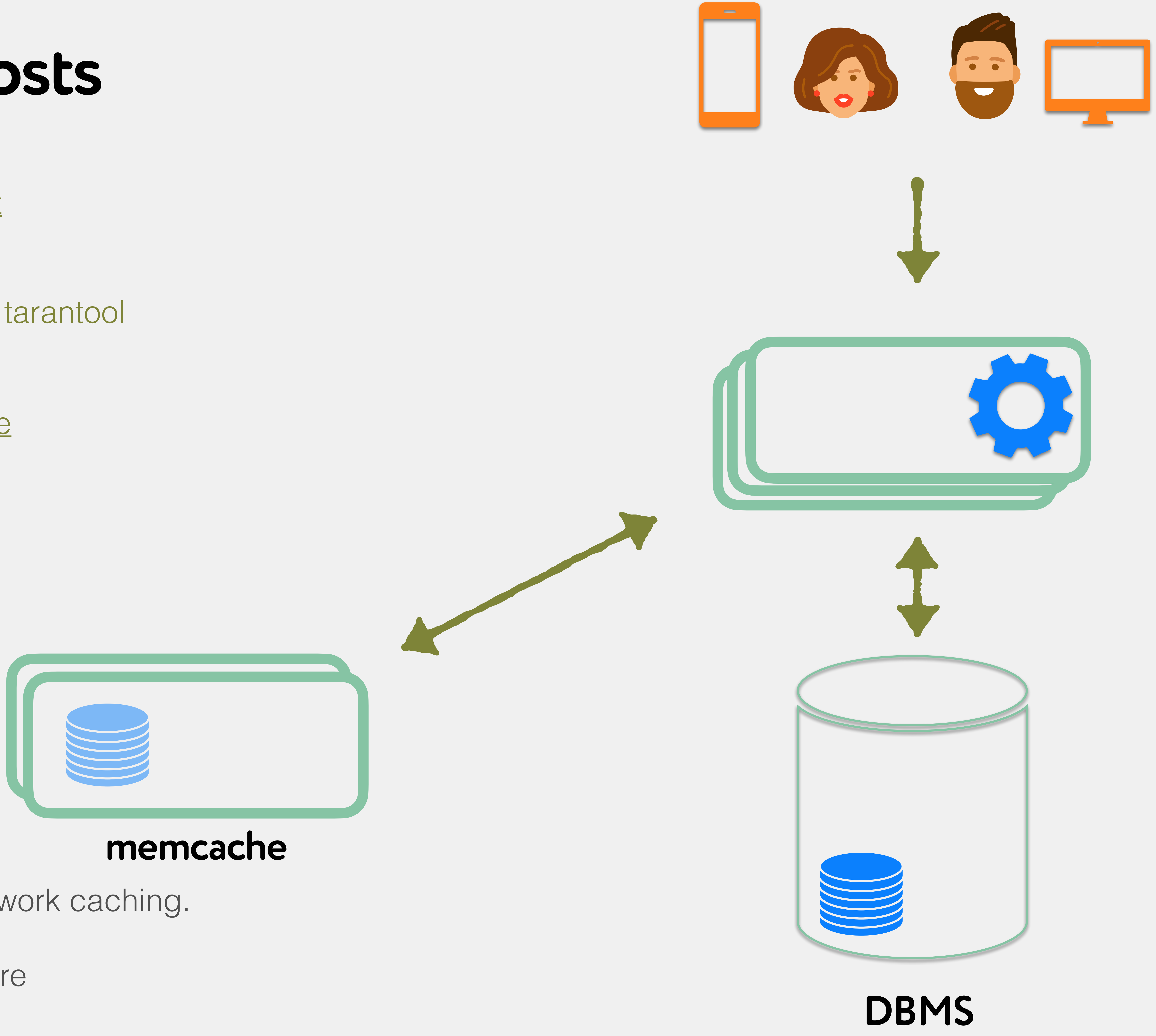
- CPU: (Un) Marshalling
- Overreads, overwrites
- Network latency and traffic



(1) Fast key-value stores: An idea whose time has come and gone
Adya et al. HotOS '19, May 13–15, 2019, Bertinoro, Italy

Microservices: lowering costs

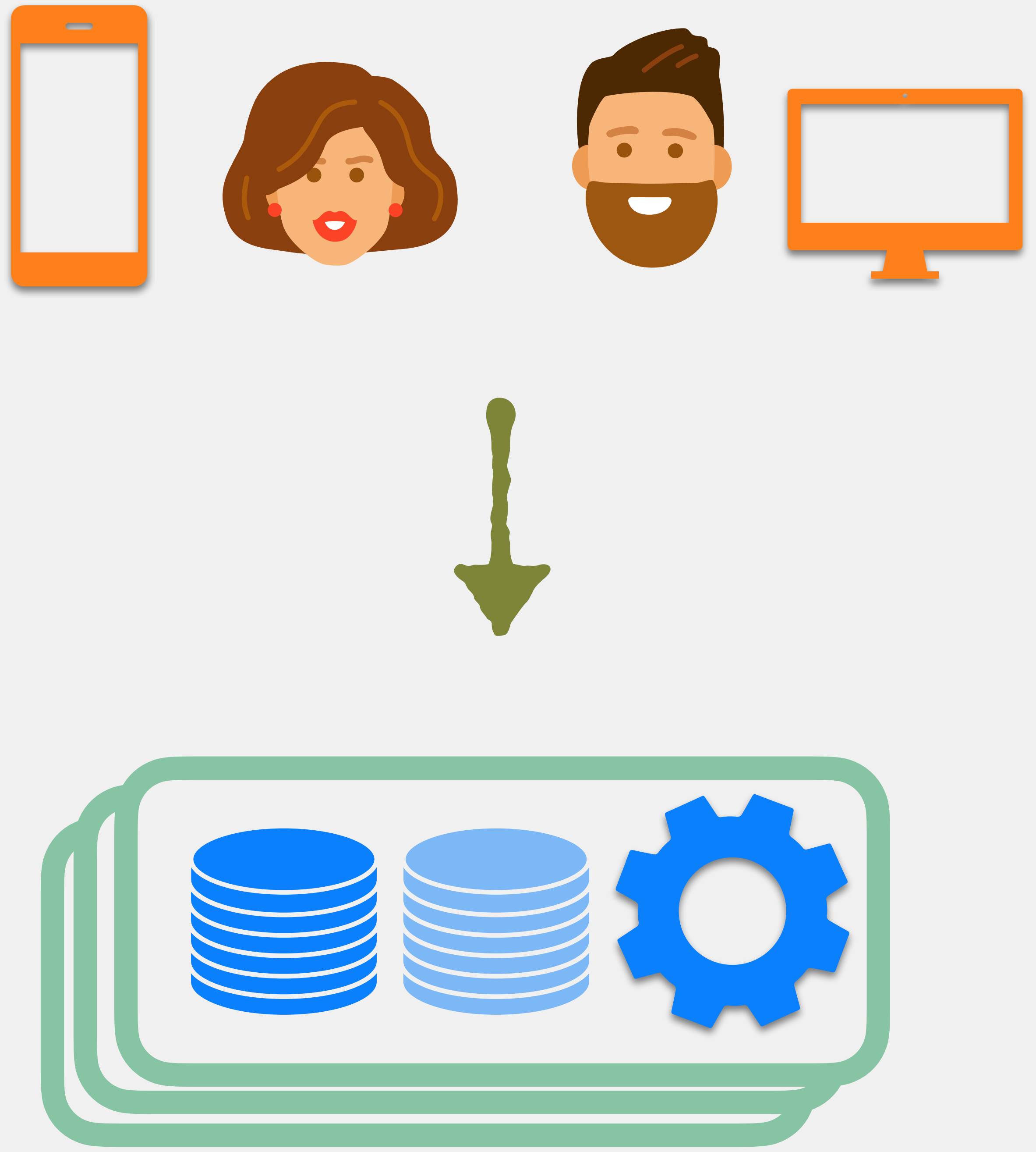
- CPU: (Un) Marshalling — (5) KV-Direct
- Overreads, overwrites — (2) redis, (3) tarantool
- Network latency and traffic — (4) NetCache



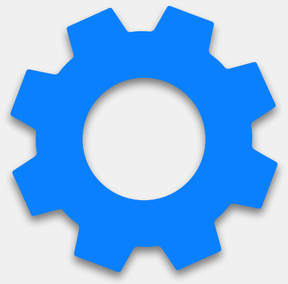
(2) http://redis.io
(3) https://tarantool.io
(4) Netcache: Balancing key-value stores with fast in-network caching. In X. Jin et al, Stoica. SOSP, 2017.
(5) Kv-direct: high-performance in-memory key-value store with programmable nic. B. Li et al, In SOSP, 2017.

Stateful Microservices

Stateful Microservices



- CPU: (Un)Marshalling
- Overreads, overwrites
- Network latency and traffic

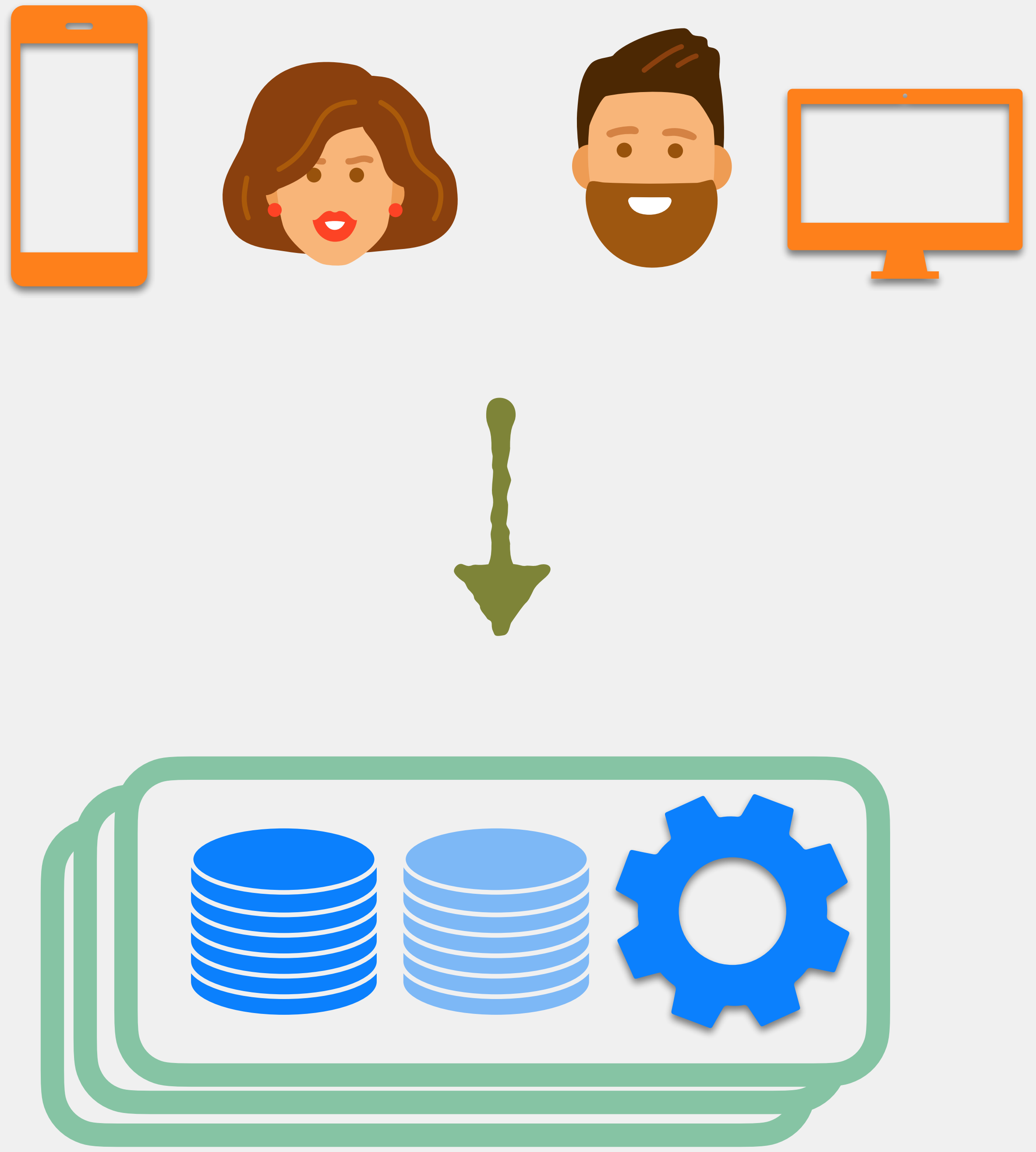


Application Logic

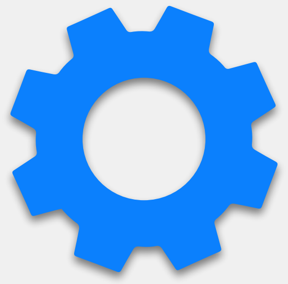
Custom in-memory store
application specific

Embedded Distributed store code
only the code is embedded,
operates just like a dedicated node

Stateful Microservices



More efficient. Ever.

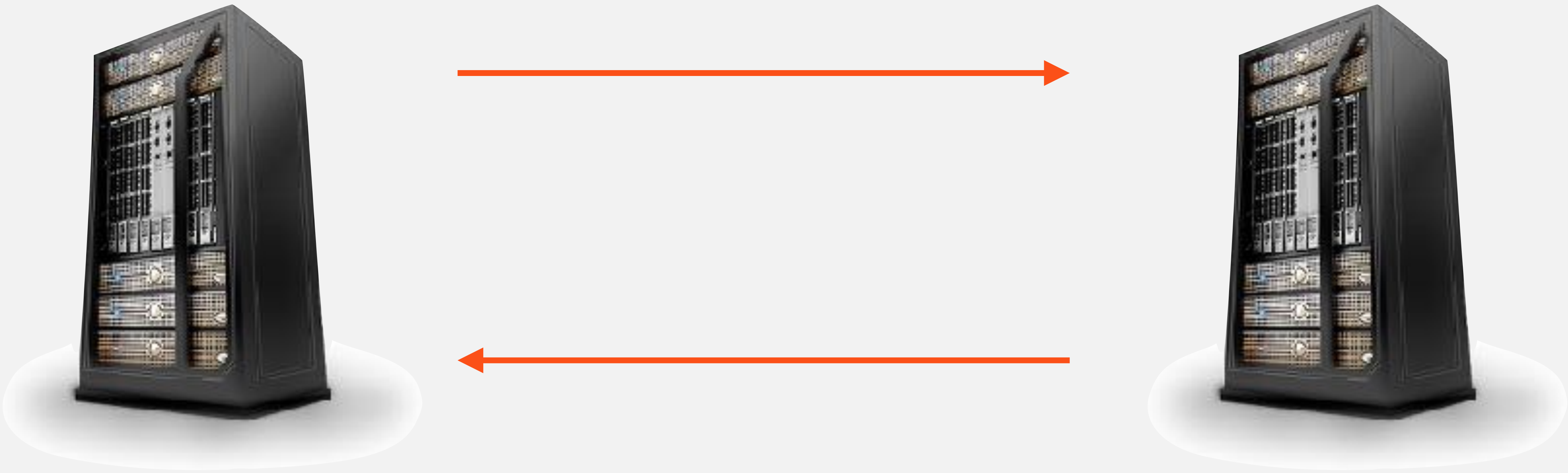


Application Logic

Custom in-memory store
application specific

Embedded Distributed store code
only the code is embedded,
operates just like a dedicated node

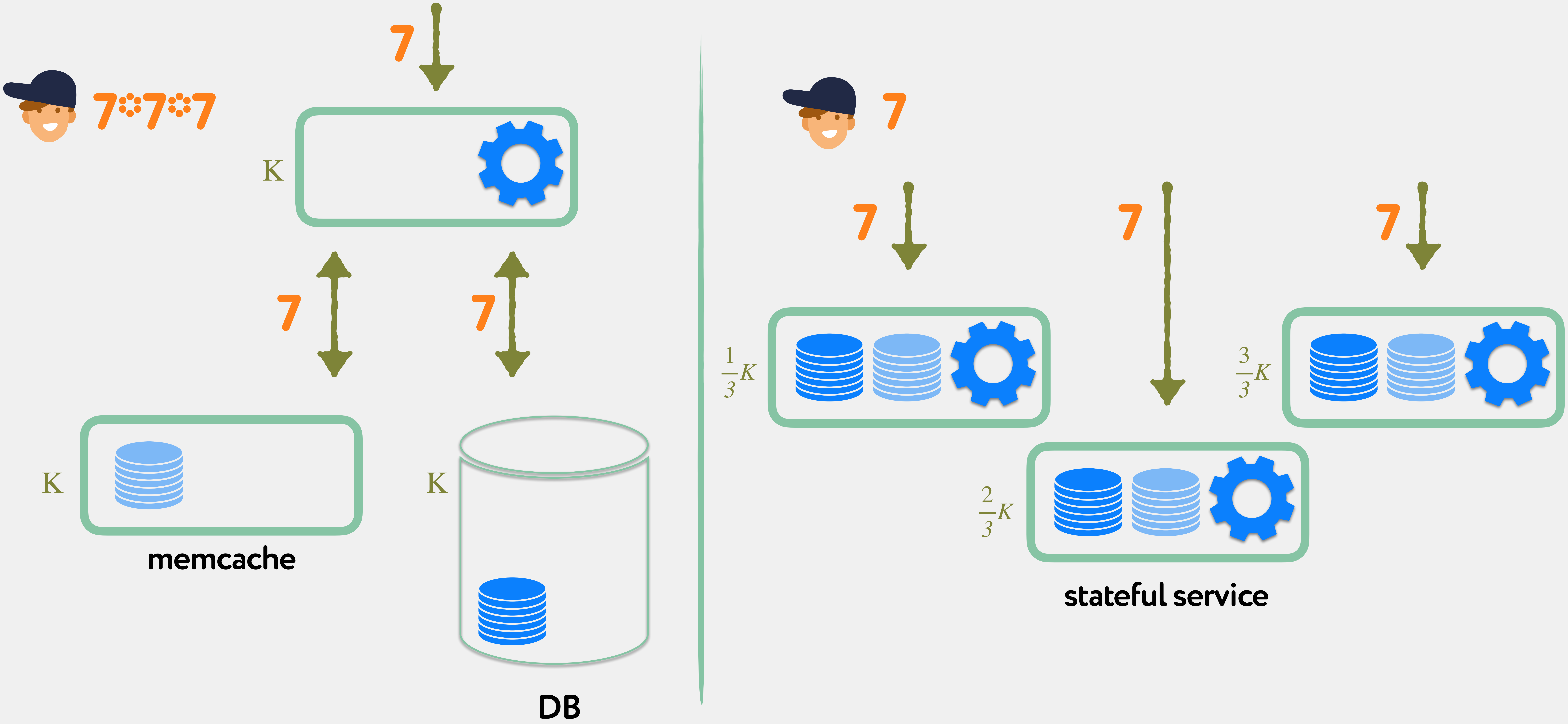
What ~~can~~ will go wrong?



1. Client crash
2. Server crash
3. Request omission
4. Response omission
5. Server Timeout
6. Invalid value response
7. Arbitrary failure

(6) Distributed systems at OK.ru
Oleg Anastasyev, Armenian C++ Community Meetup #7 @ ISTC, on 17.12.2022
http://t.me/cpparm

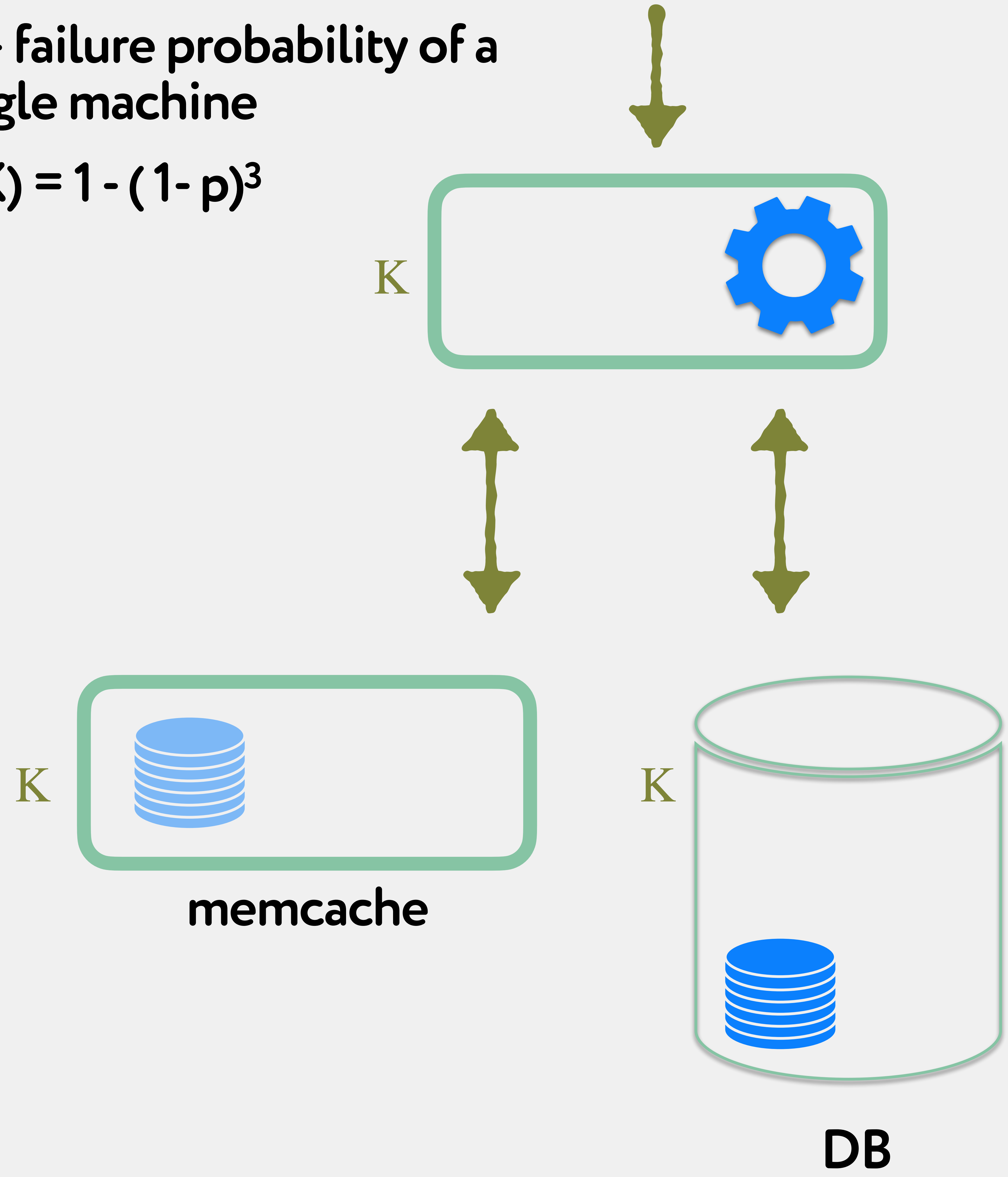
Failures


7 7 7
7
K
7
7
K
memcache
K
DB
7
7
7
7
$\frac{1}{3}K$
$\frac{3}{3}K$
$\frac{2}{3}K$
stateful service

Downtime probabilities

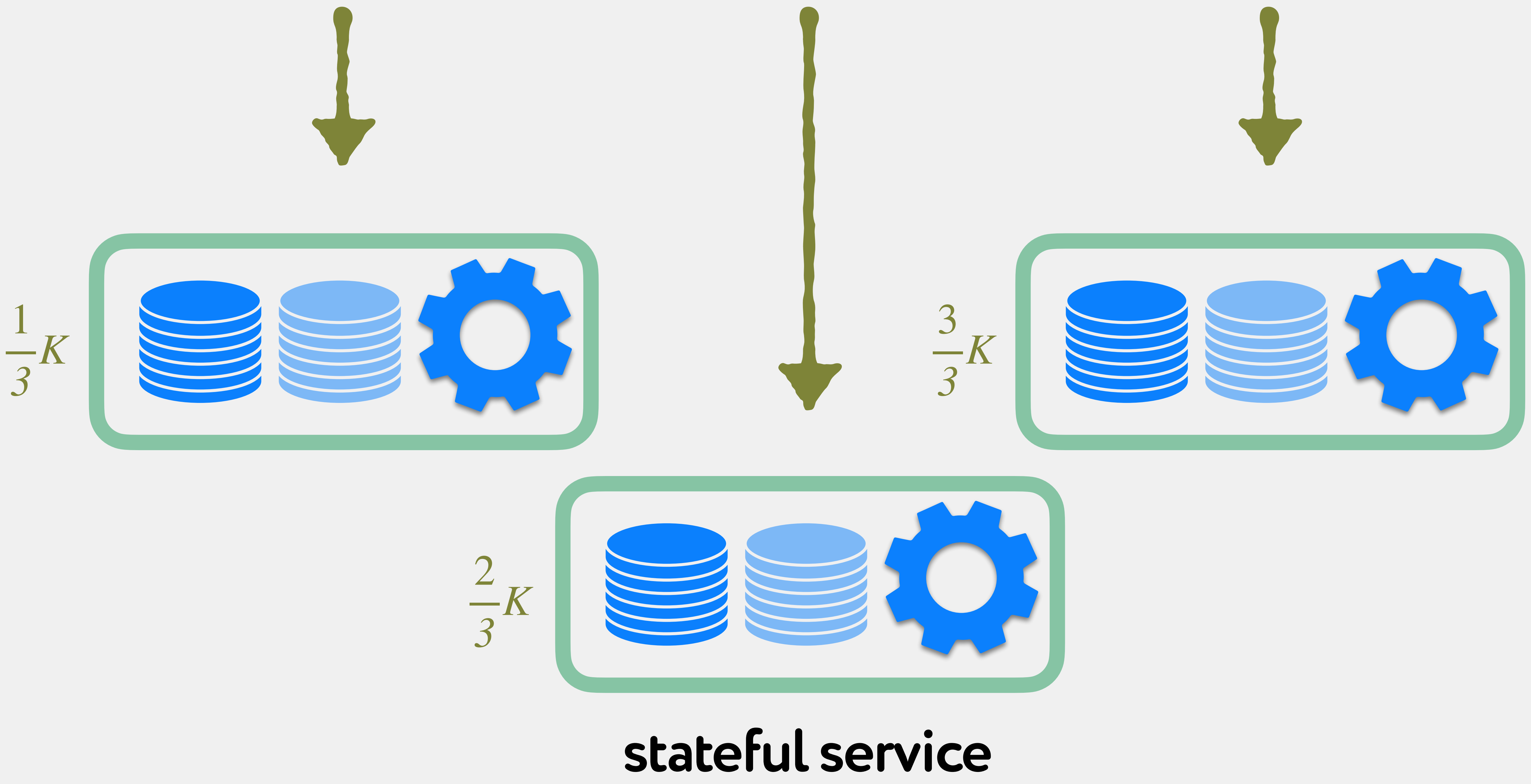
p — failure probability of a single machine

$P(K) = 1 - (1 - p)^3$



$P(K) = p^3$

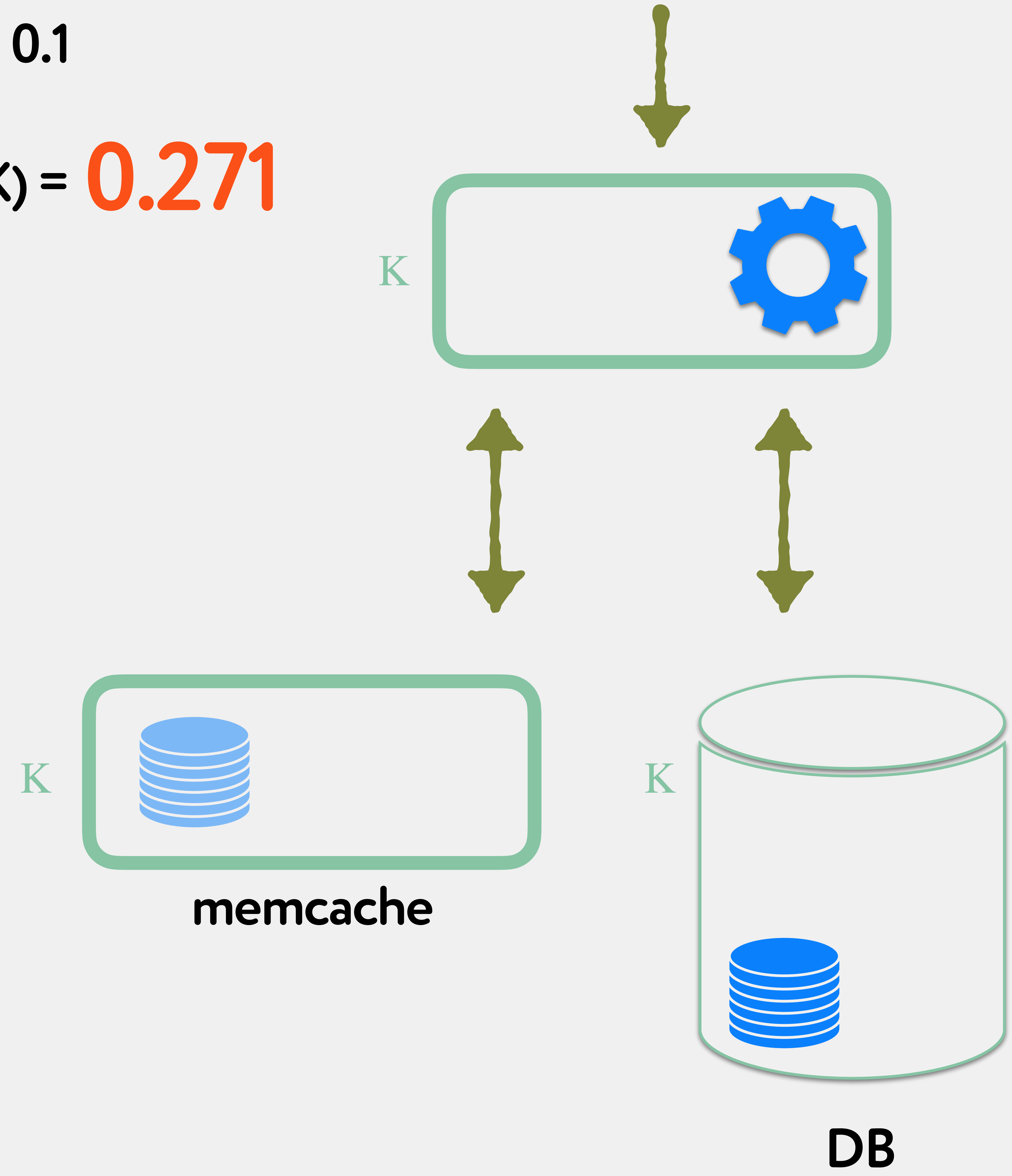
$P(⅓\,K) = 1 - (1 - p)^3$



Downtime probabilities

$p = 0.1$

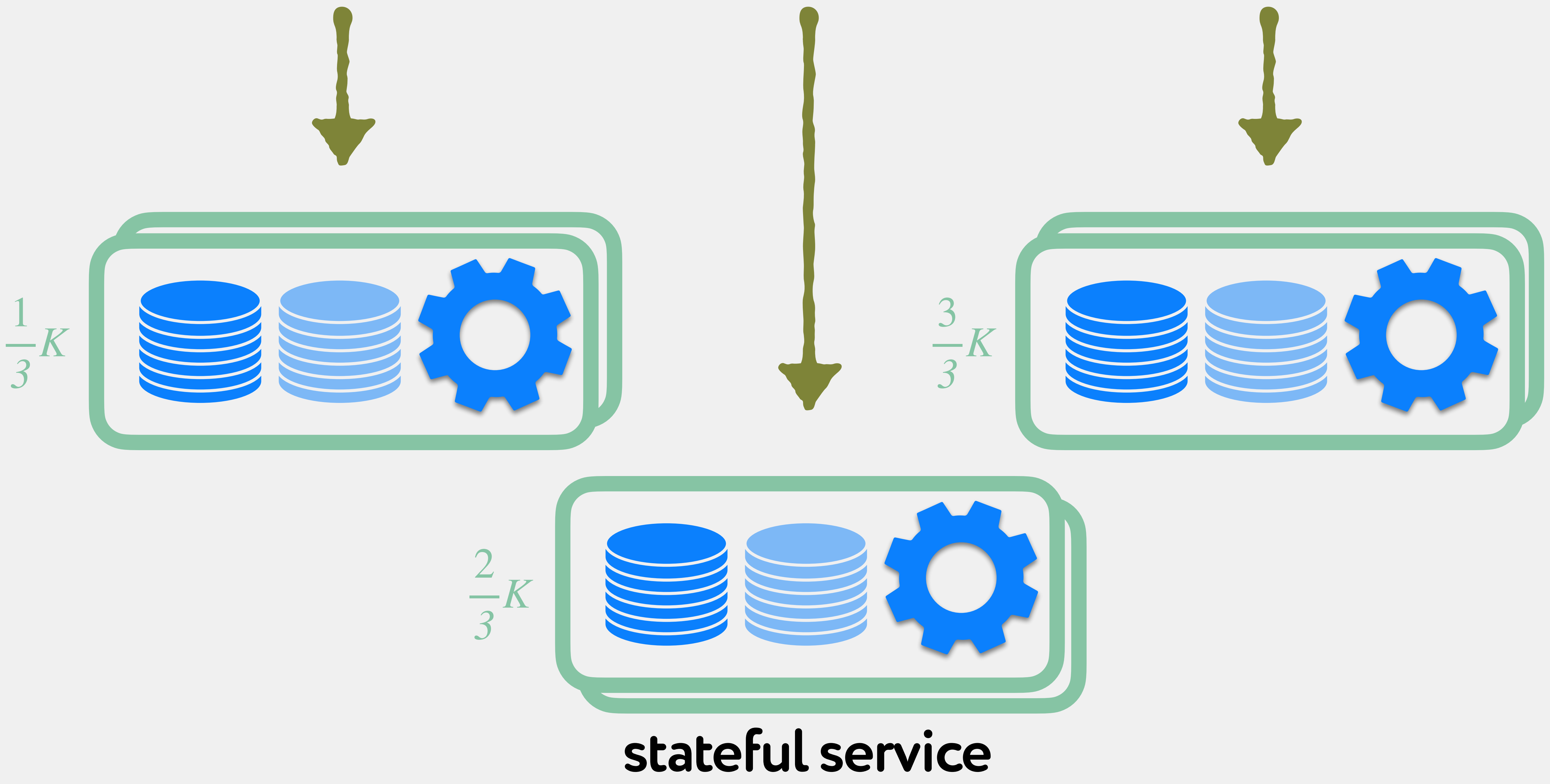
$P(K) = 0.271$



$P(K) = 0.001$

$P(\frac{1}{3} K) = 0.271$

More reliable



2 Implementation

Embedding the Database

requirements are:

- **Always available**

  Replication, Consistency

- **Scalable**

  Re-sharding

- **Application language**

  Minimal (un) marshalling,
  Integration with the application

- **Open Source**

  so we can code something crazy

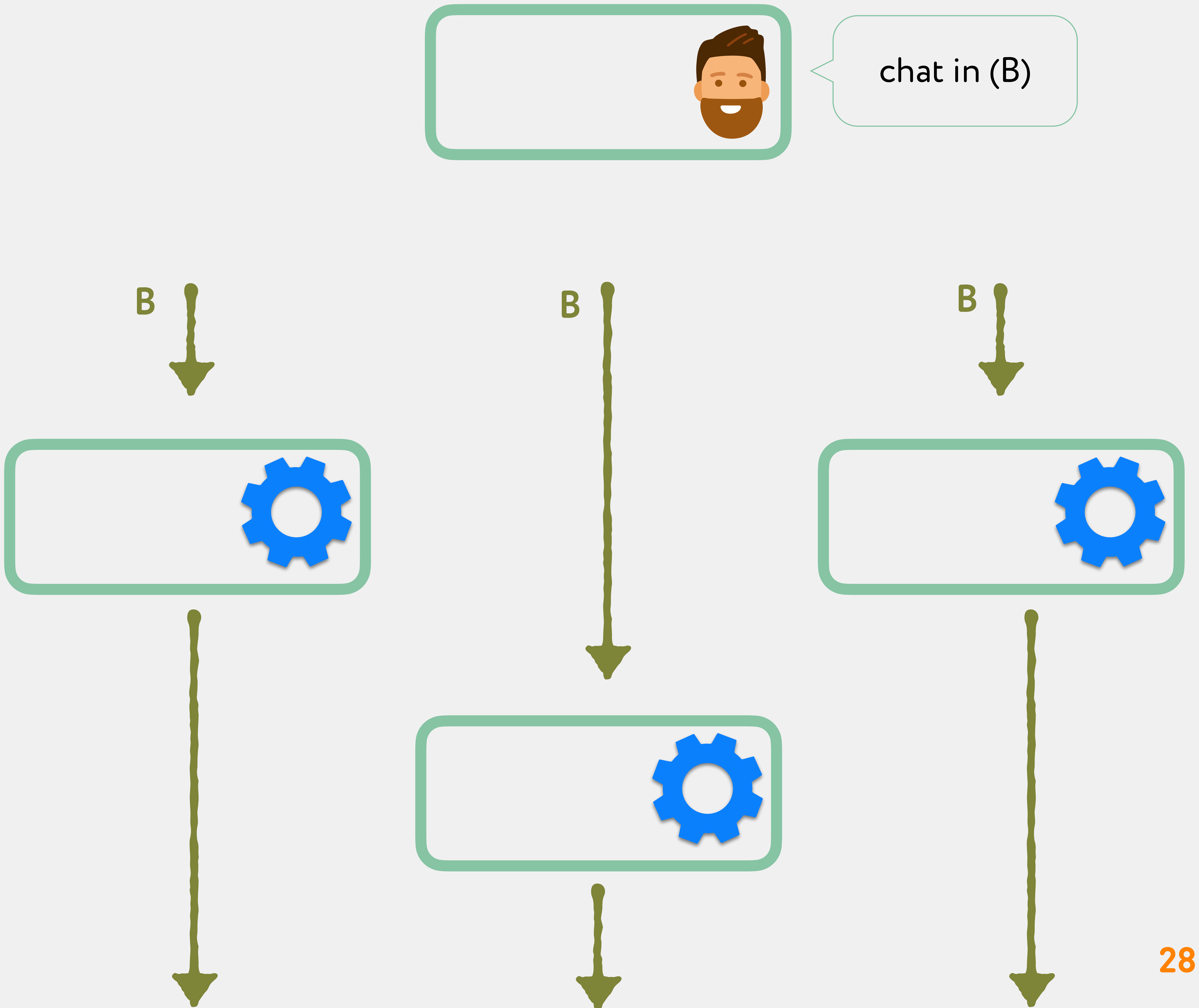
Embedding the Database

```java
package org.apache.cassandra.service;

public class CassandraDaemon
{
```

1. -cp cassandra/lib/*.jar

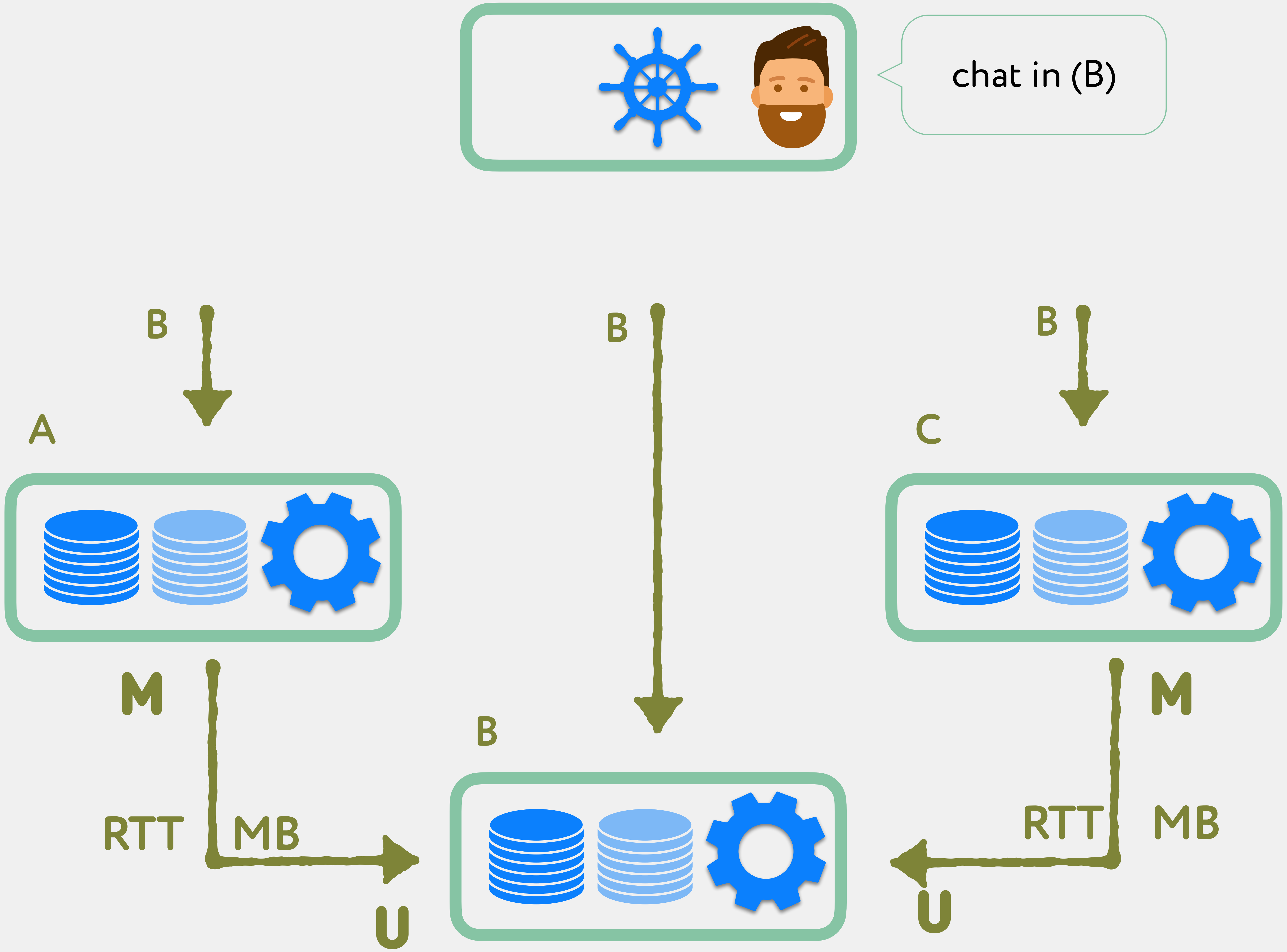
2.
```java
System.setProperty( "cassandra.config", "file://whatever/cassandra.yaml" );

CassandraDaemon.instance.activate();
```

Request routing



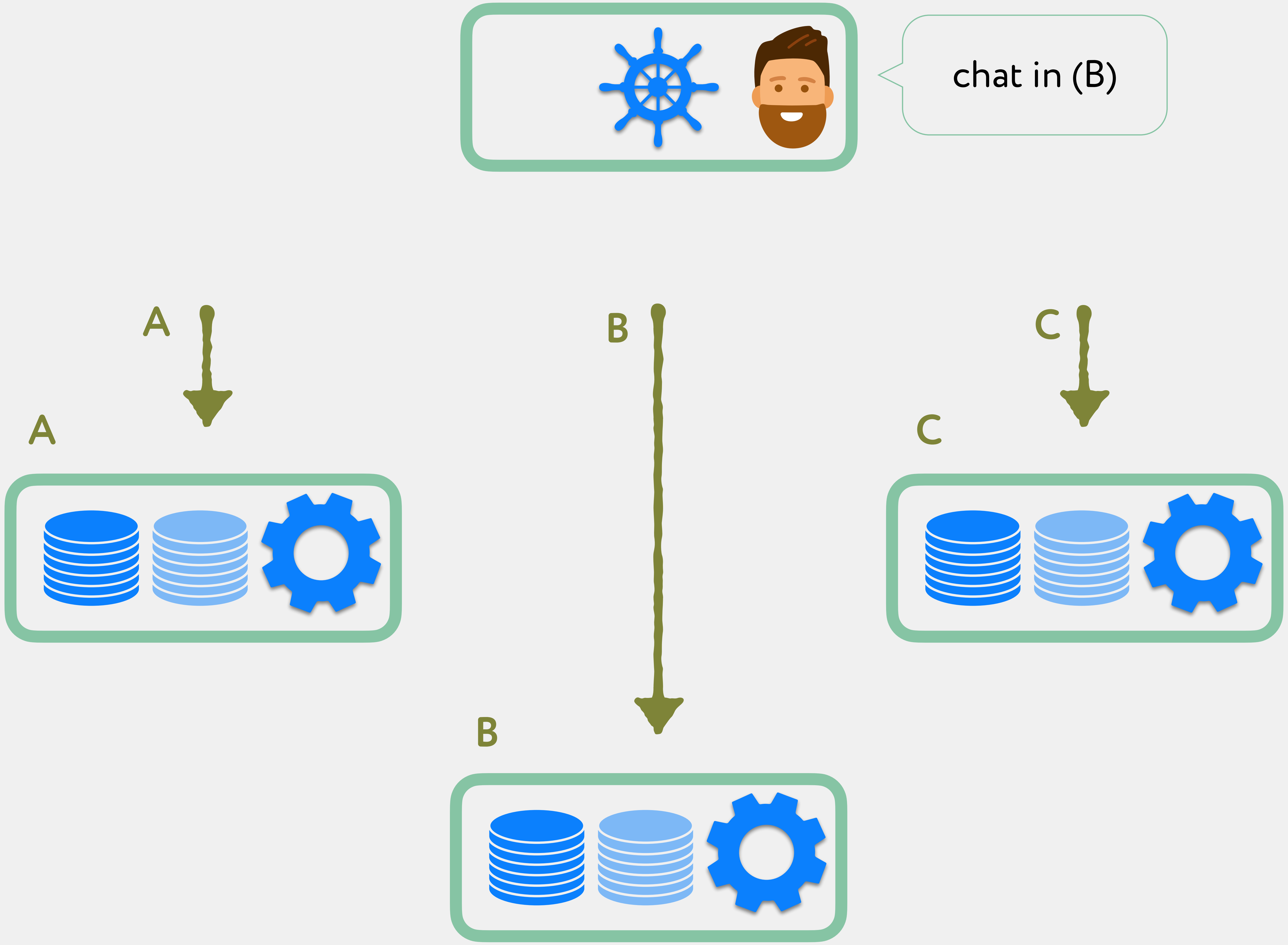
Request routing

- **Partition-aware client routing library**
  Routes request to the replica owning the data, based on the key specified in a request and the cluster topology information



Request routing

- **Partition-aware client routing library**
  Routes request to the replica owning the data, based on the key specified in a request and the cluster topology information



Data partitioning

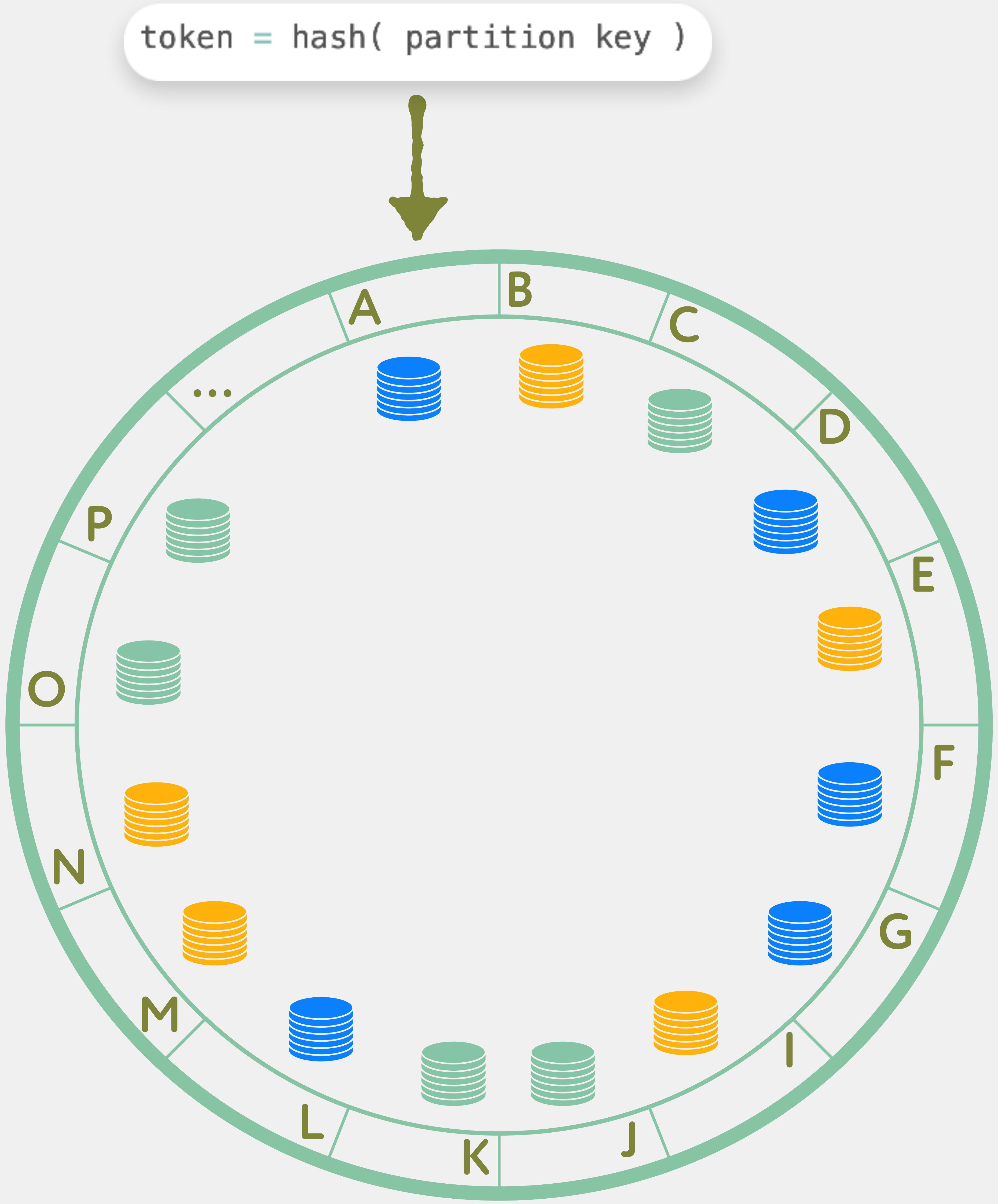
- **Partition Key (chatId)**
  Defines which node owns a row

- **Clustering Key (msgId)**
  Defines an order of rows within a partition

```
CREATE TABLE Messages (
    chatId, msgId

    user, type, text, attachments[], terminal, deletedBy[], replyT

    PRIMARY KEY ( chatId, msgId )
)
```

Data partitioning

- **Partitioner**
  Calculates token and position on ring
- **TokenMetadata**
  Maps range of tokens to primary nodes
- **Replication Strategy**
  Defines replica placement



Data partitioning

- **Partitioner**
  Calculates token and position on ring
- **TokenMetadata**
  Maps range of tokens to primary nodes
- **Replication Strategy**
  Defines replica placement

```
SortedMap<Token, List<InetAddress>> endpointMap = ...

AbstractReplicationStrategy replication = ...

for ( Token token : tokenMetadata.sortedTokens() ) {
    endpointMap.put( token,  replication.getNaturalEndpoints( token ) );
}
```

+ **Topology changes over the time**
  Refreshes and dealing with stale topology

Messenger: calling the DB

```
CREATE TABLE Messages (
    chatId, msgId

    user, type, text, attachments[], terminal, delete

    PRIMARY KEY ( chatId, msgId )
)
```

- getMessages(viewer, chat, from, to)

Quick start

Here's a short program that connects to Cassandra and executes a query:

```
import com.datastax.oss.driver.api.core.CqlSession;
import com.datastax.oss.driver.api.core.cql.*;

try (CqlSession session = CqlSession.builder().build()) {                    // (1)
  ResultSet rs = session.execute("select release_version from system.local");  // (2)
  Row row = rs.one();
  System.out.println(row.getString("release_version"));                     // (3)
}
```

(7) https://github.com/datastax/java-driver/tree/4.x/manual/core

Messenger: calling the DB

```
CREATE TABLE Messages (
    chatId, msgId

    user, type, text, attachments[], terminal, delete

    PRIMARY KEY ( chatId, msgId )
)
```

- getMessages(viewer, chat, from, to)
- add(chat, message)

```
package org.apache.cassandra.cql3;

import java.nio.ByteBuffer;

public class QueryProcessor
{
    public static UntypedResultSet execute(String query,
                                           ConsistencyLevel cl, Object... values)
                                           throws RequestExecutionException
```

```
UntypedResultSet rs = QueryProcessor.execute(
                "SELECT * FROM Messages "
                + "WHERE chatId = ? AND msgId < ? AND msgId > ?",
                ConsistencyLevel.QUORUM, chatId, from, to );

rs.forEach( row -> {} );
```

Messages In-Memory Store

600 bi
messages

5 bi
chats

100 TB

5%
chats are active

80%
freshest 13 messages

in-memory stored data

3+ bi
messages

250 mi
chats

500 GB

Messages In-Memory Store

- getMessages(viewer, chat, from, to)
- getLastMessages(viewer, chats)
- add(chat, message)

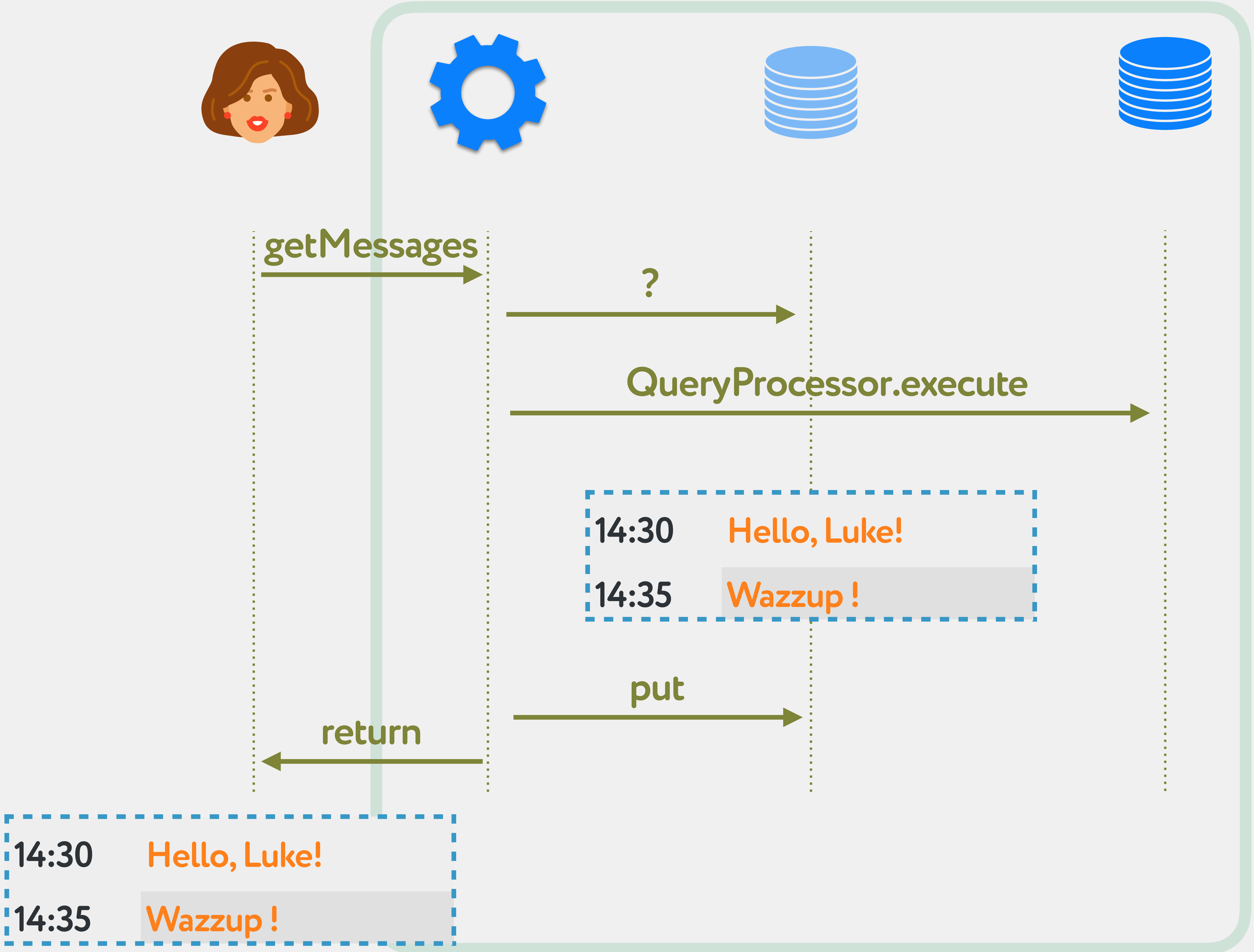
in-memory store

3+ bi
messages

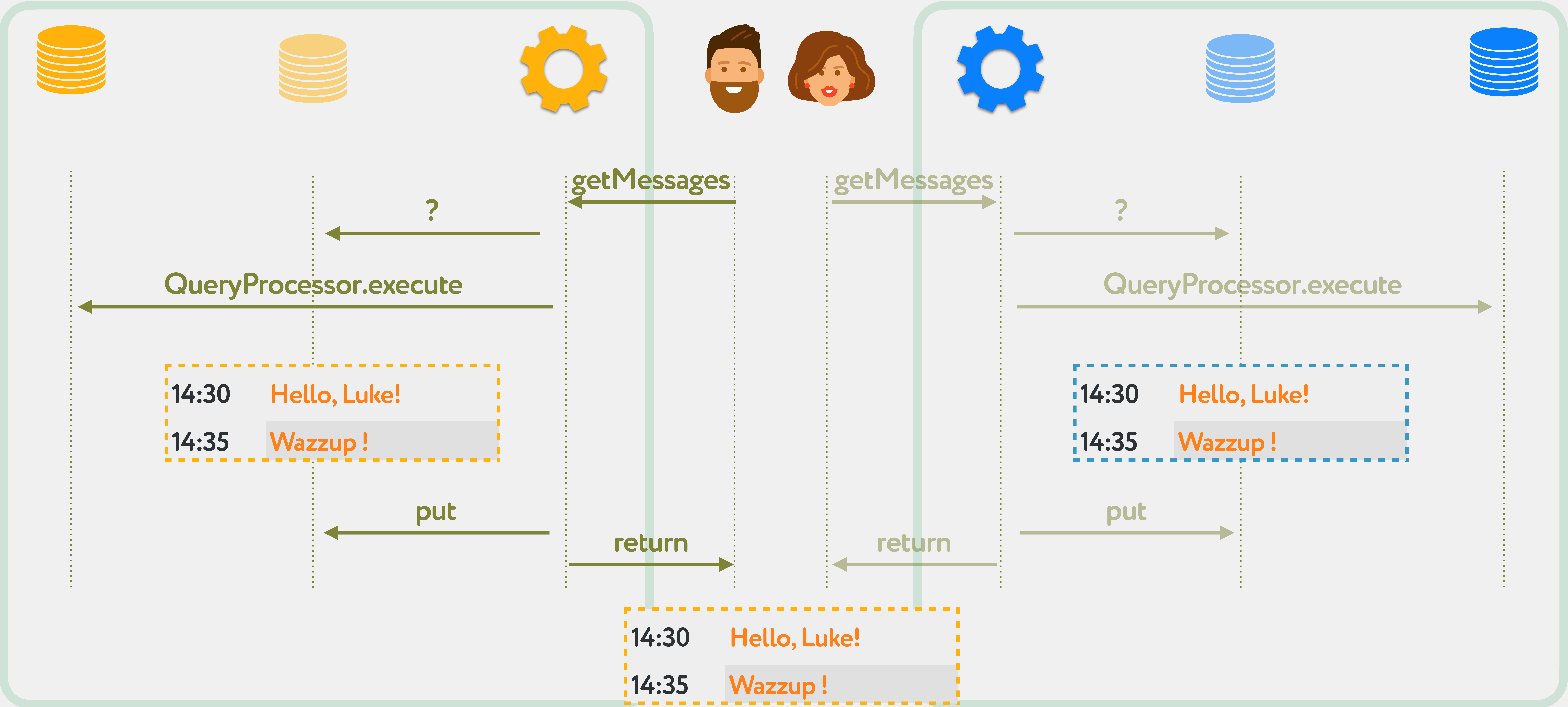
250 mi
chats

500 GB

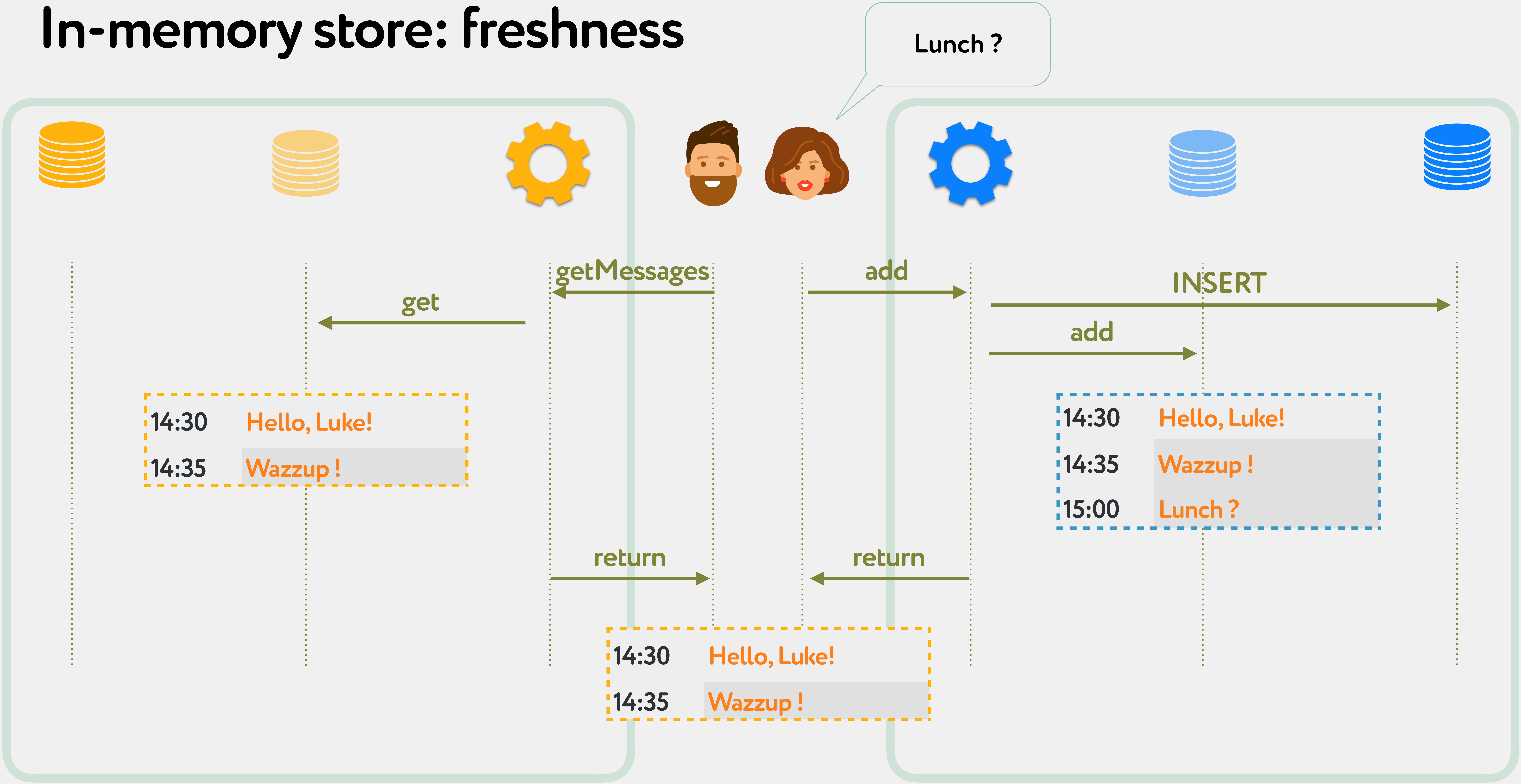
Messages In-Memory Store: getMessages



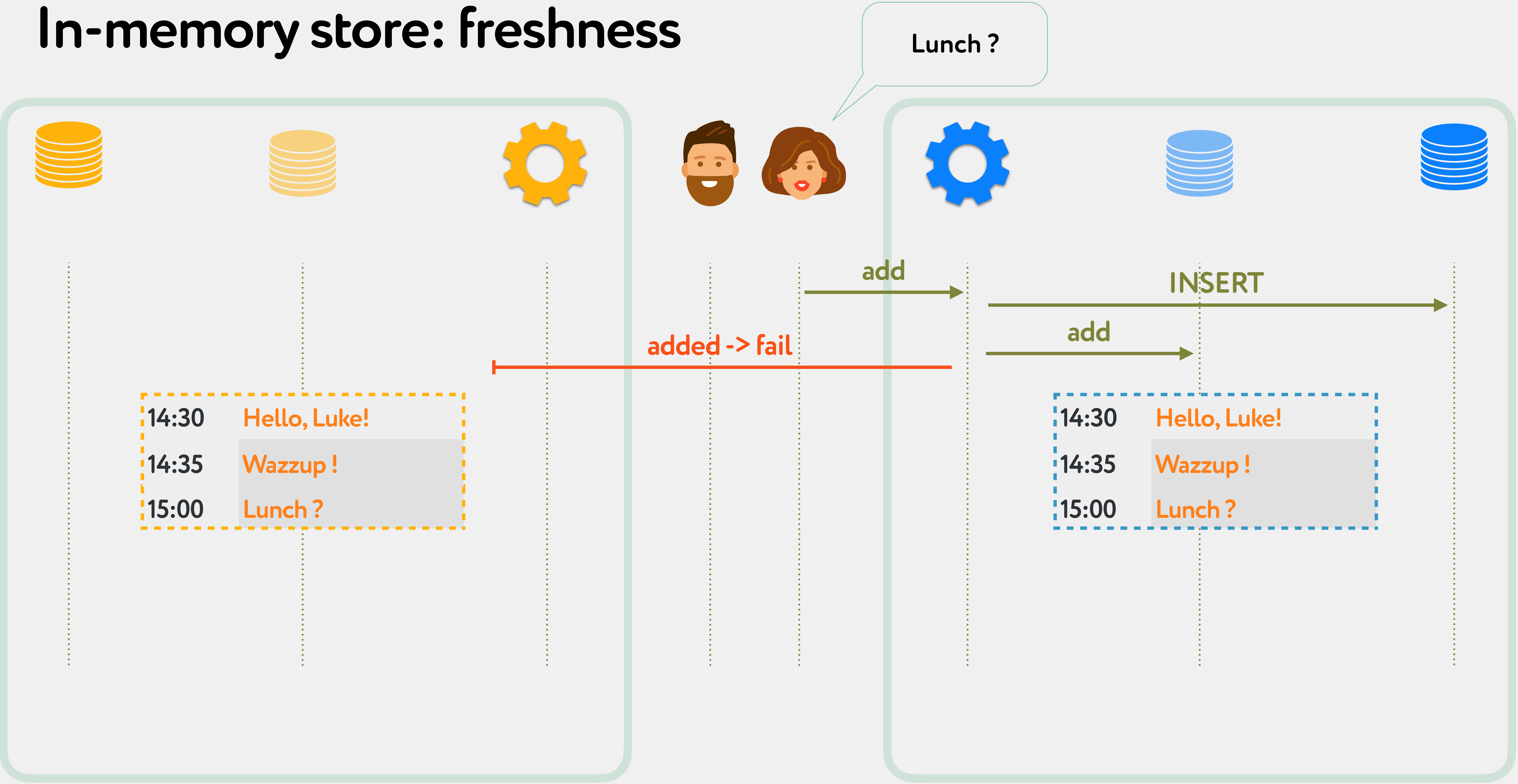
Messages In-Memory Store



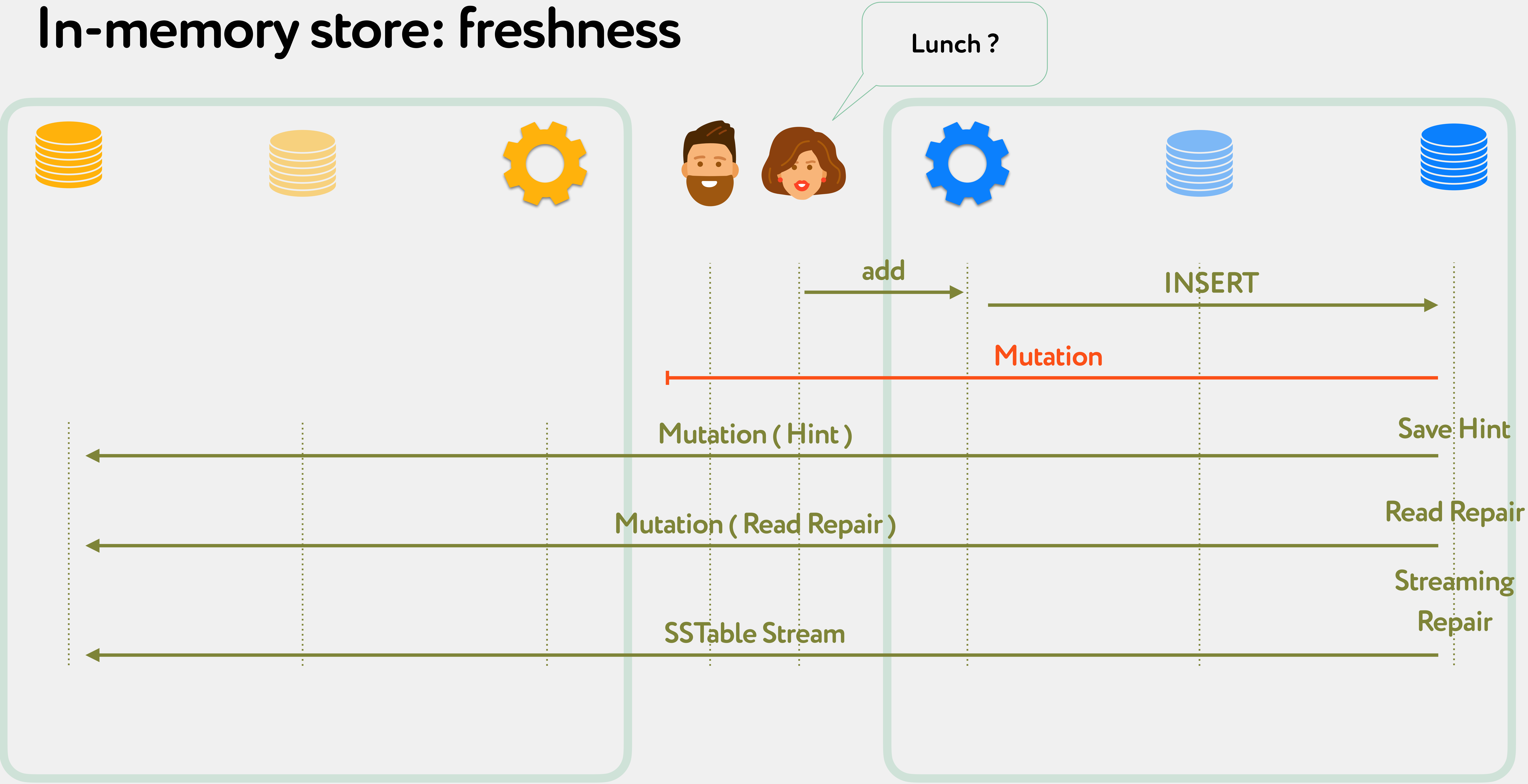
In-memory store: freshness



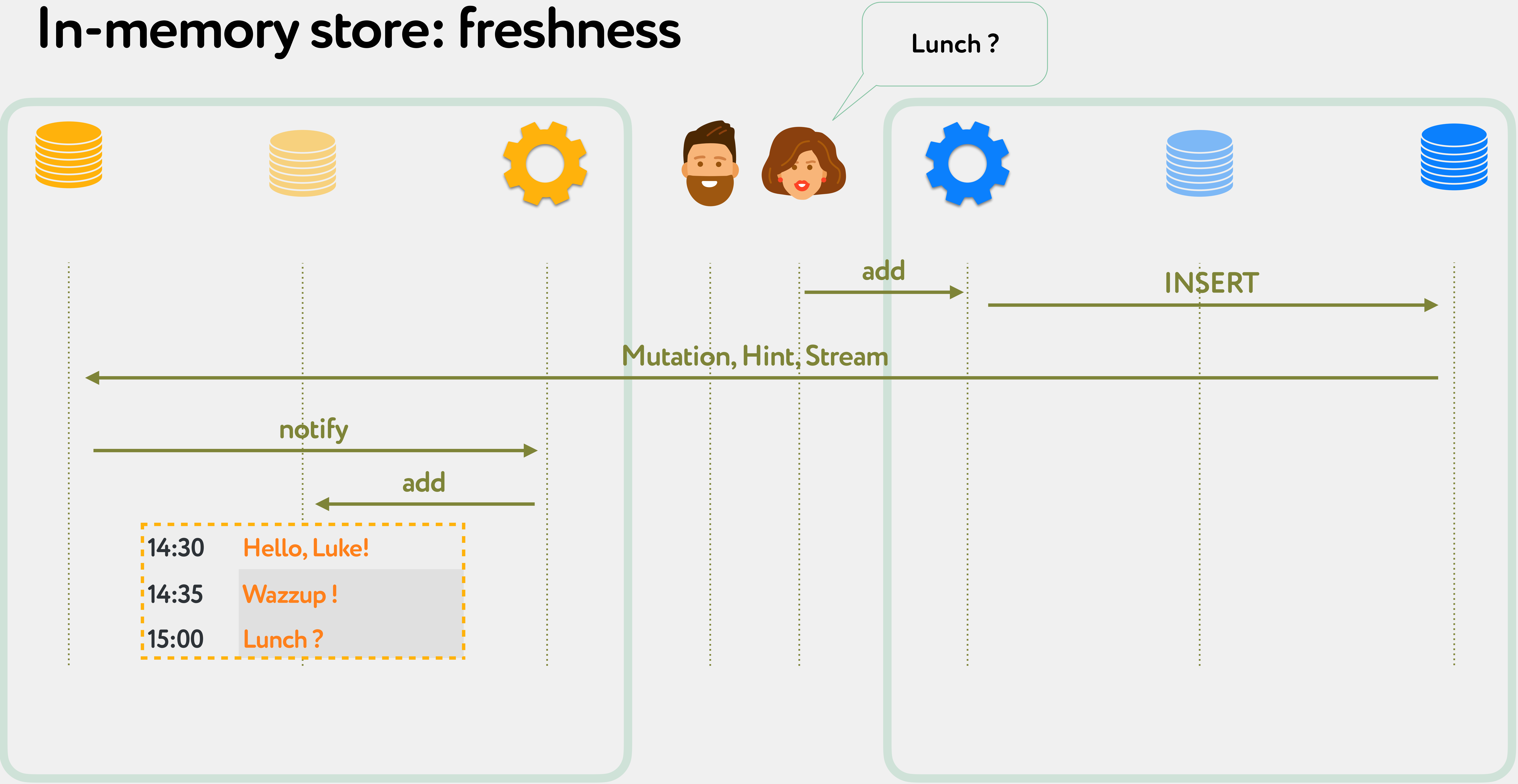
In-memory store: freshness



In-memory store: freshness



In-memory store: freshness



Mutation — individual rows:

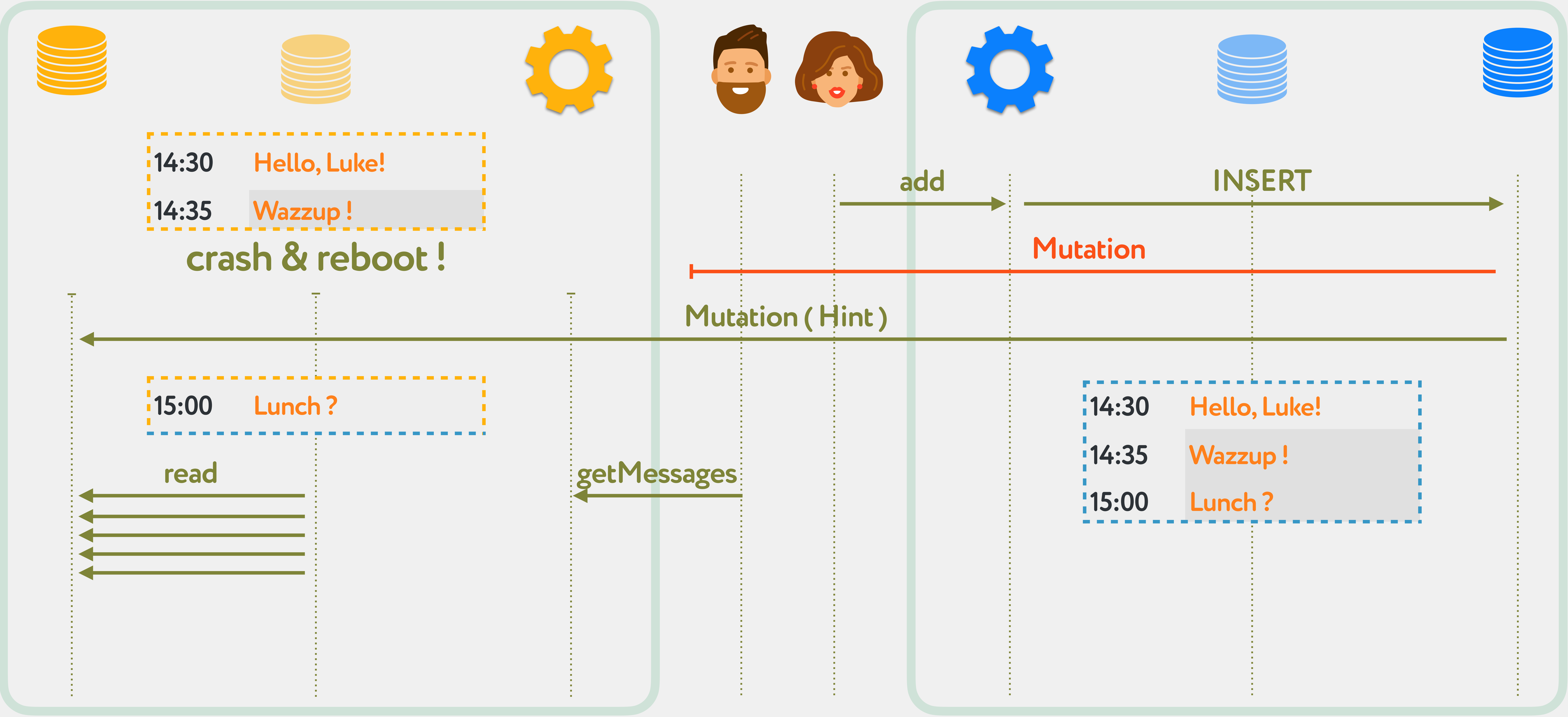
```
interface ApplyMutationListener
{
    void onApply(ByteBuffer key,
                 DeletionTime deletion,
                 Iterator<Unfiltered> atoms);
}
```

```
package org.apache.cassandra.db;

public class Keyspace
{
    public void apply( Mutation mutation,
                       boolean writeCommitLog,
                       boolean updateIndexes,
                       boolean isDroppable )
```

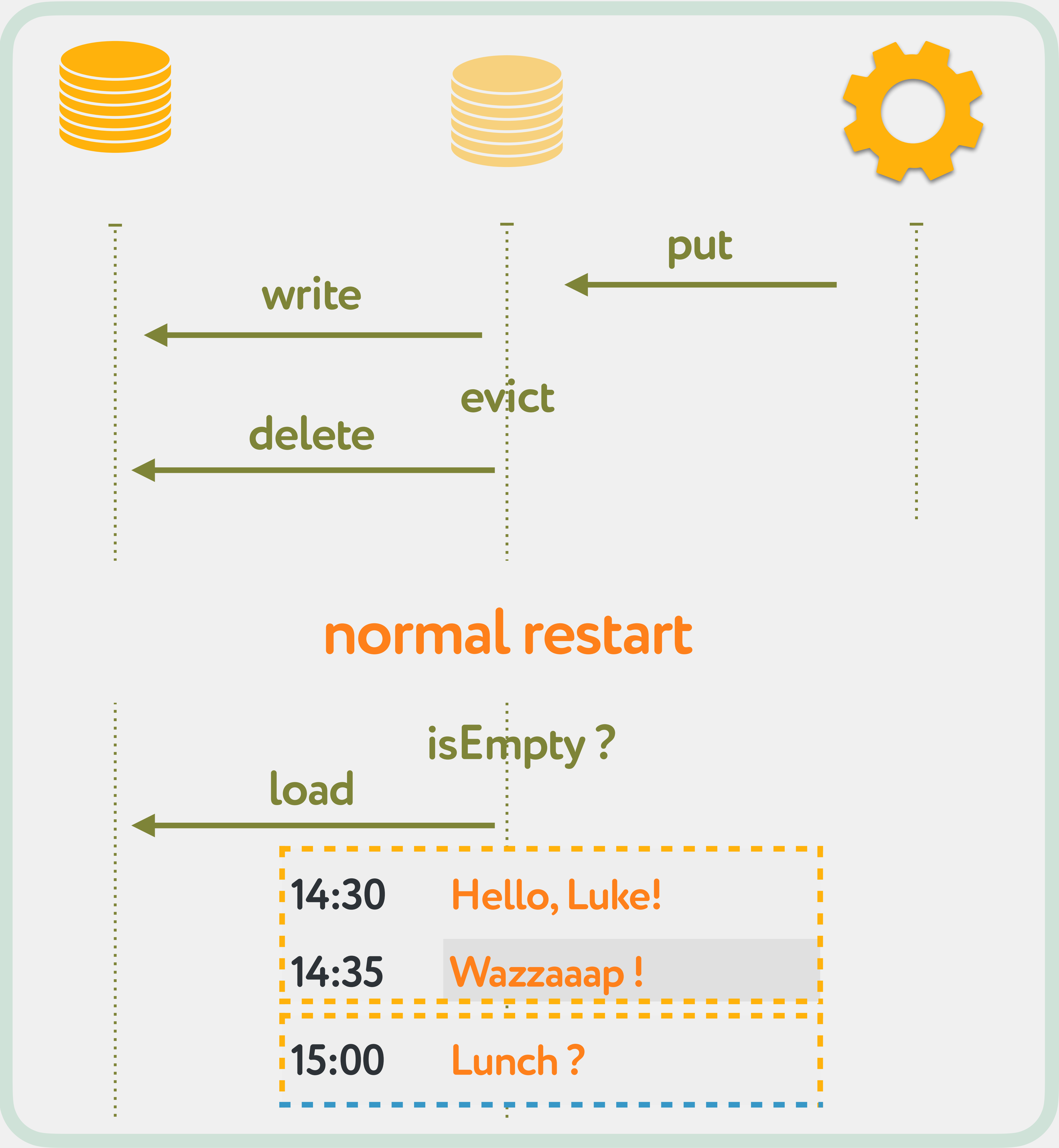
Streaming — ss tables:

```
package org.apache.cassandra.streaming;

public class StreamReceiveTask extends StreamTask
{
    //  holds references to SSTables received
    protected Collection<SSTableReader> sstables;

    private static class OnCompletionRunnable implements Runnable {
```

Messages In-Memory Store: state loss



Messages In-Memory Store: state loss




```
CREATE KEYSPACE Caches
    WITH REPLICATION = {
    'class': 'LocalStrategy'
    }
```

```
CREATE TABLE Caches.MessagesSnapshot (
    rowkey blob,
    value blob,
    PRIMARY KEY ( rowkey )
)
```

```
SELECT * FROM MessagesSnapshot
```

In-memory store: optimizing normal restarts

- Shared Memory

https://github.com/odnoklassniki/one-nio

- /dev/shm/msgs-cache.mem

  sometimes, not always

one.nio.mem

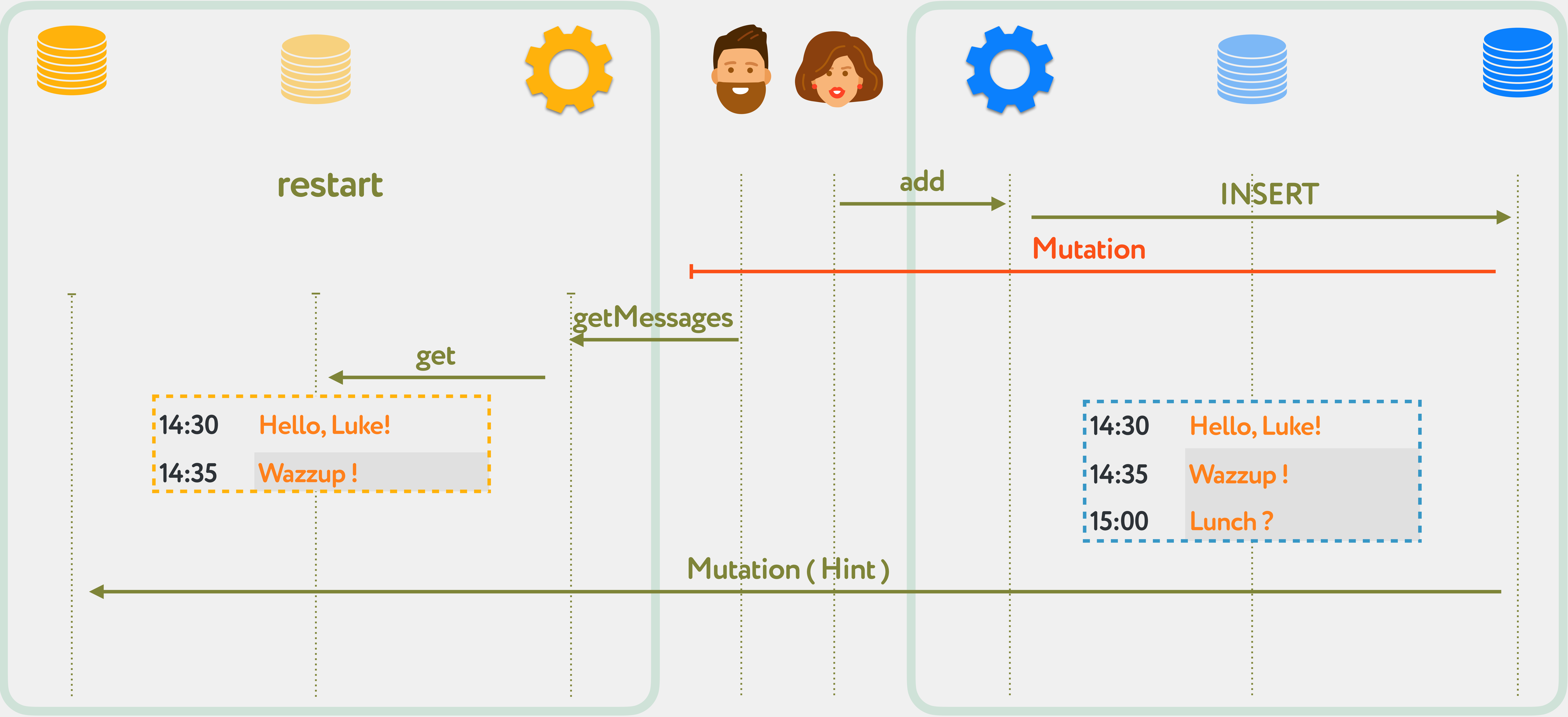
- tmpfs

SharedMemoryMap

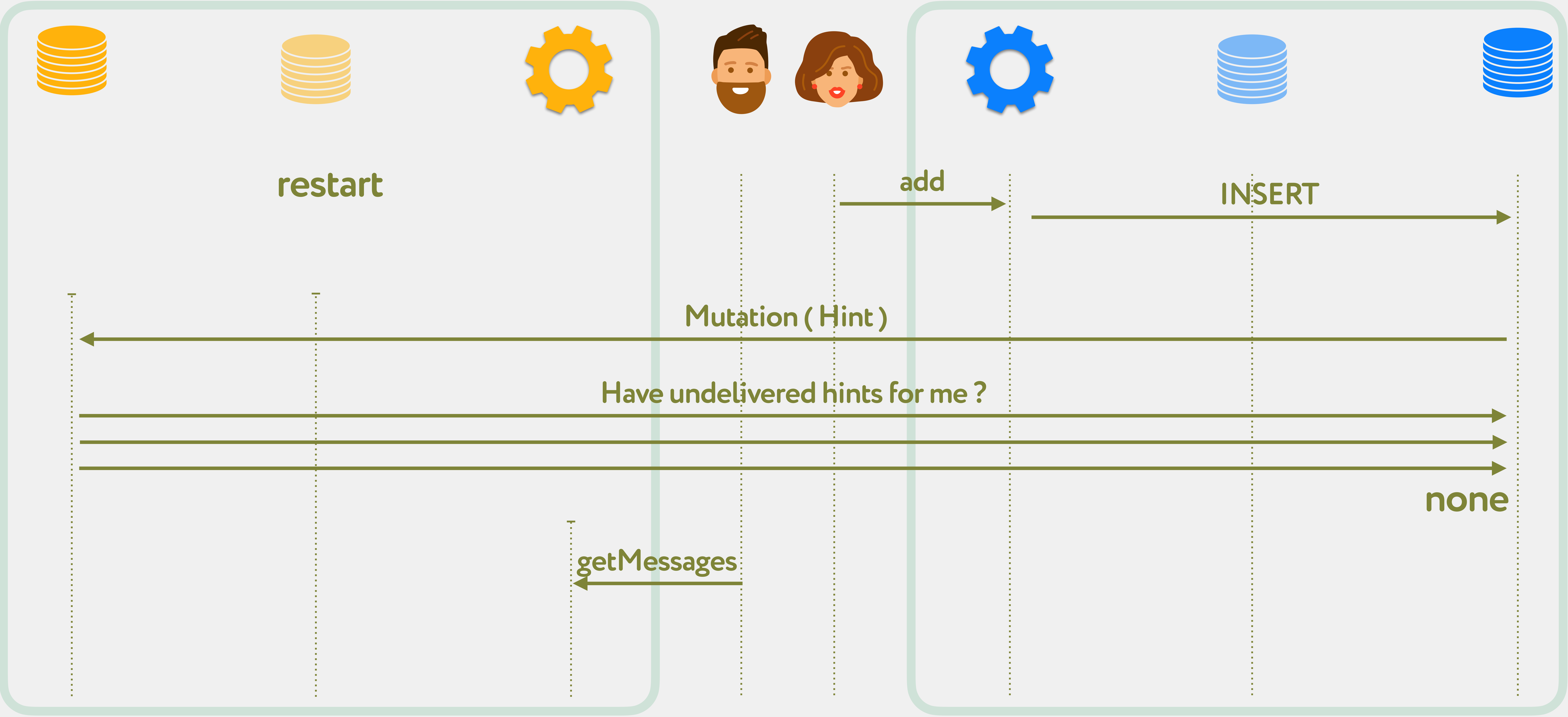
- hugetlbfs

  4K pages -> 2M, 1G

In-memory store: waiting for consistency



In-memory store: waiting for consistency



In-memory store: summary

- **Shares process with the app and the database**

  avoid marshalling and network costs, overreads

- **Data freshness problem**

  Extended Cassandra with Listeners

- **State loss problem**

  Use local tables and shared memory

- **Consistent**

  as much as the database

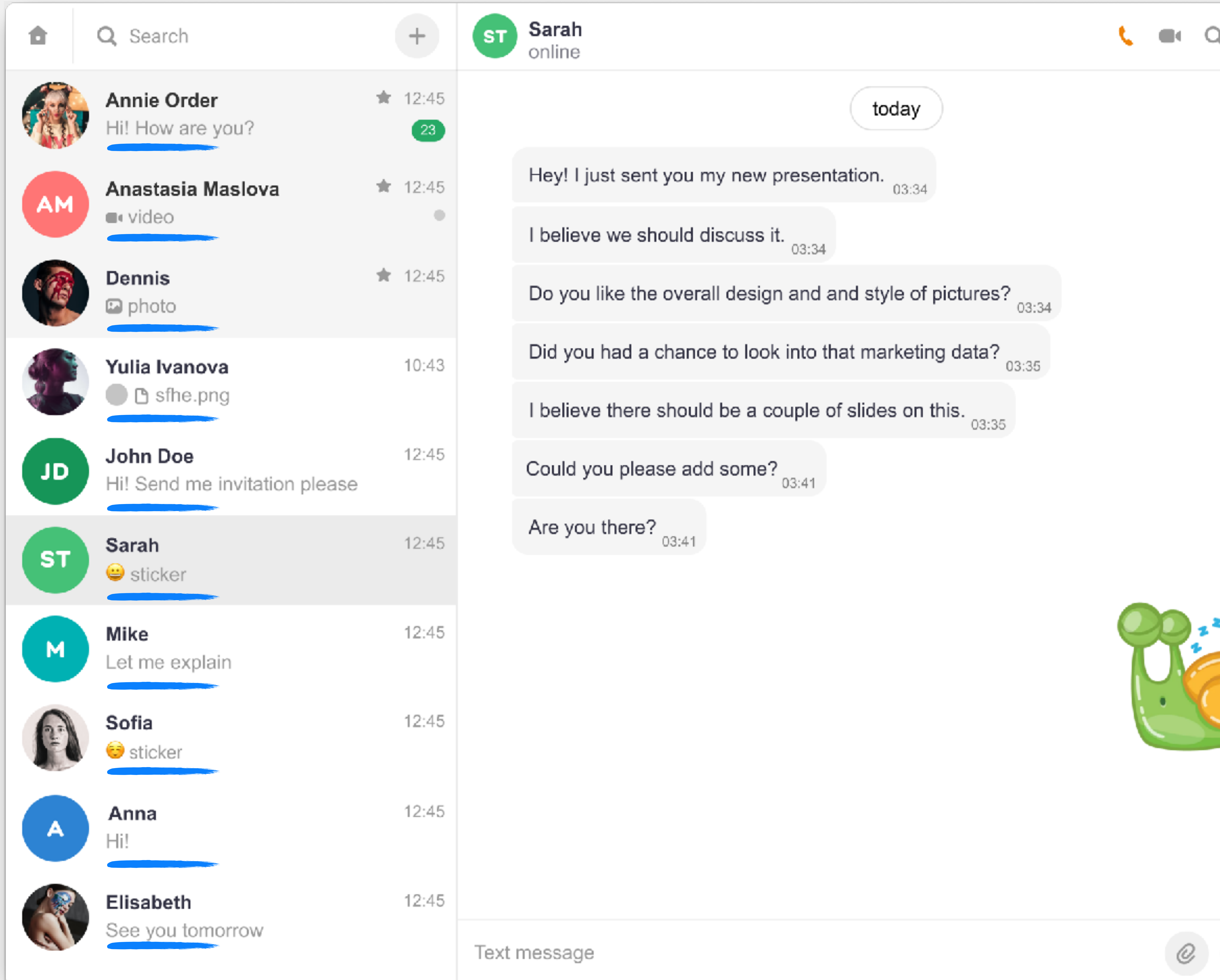
getLastMessages(chats[])

- **Multiple chats in request**

  No single node owns all data

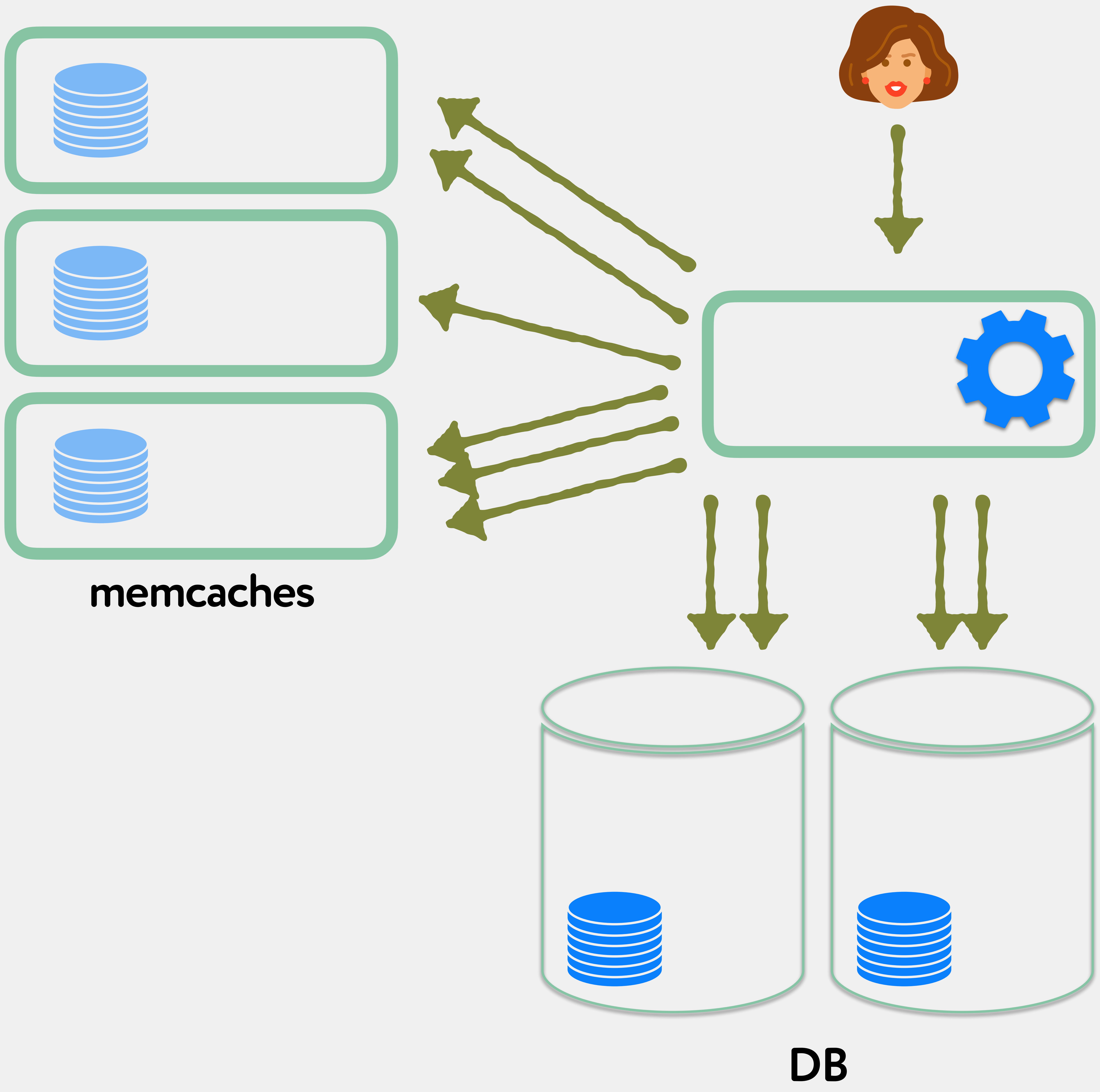
- **Fraction of them are in cache**

  Meaningless to load inactive chats to cache



getLastMessages(chats[])

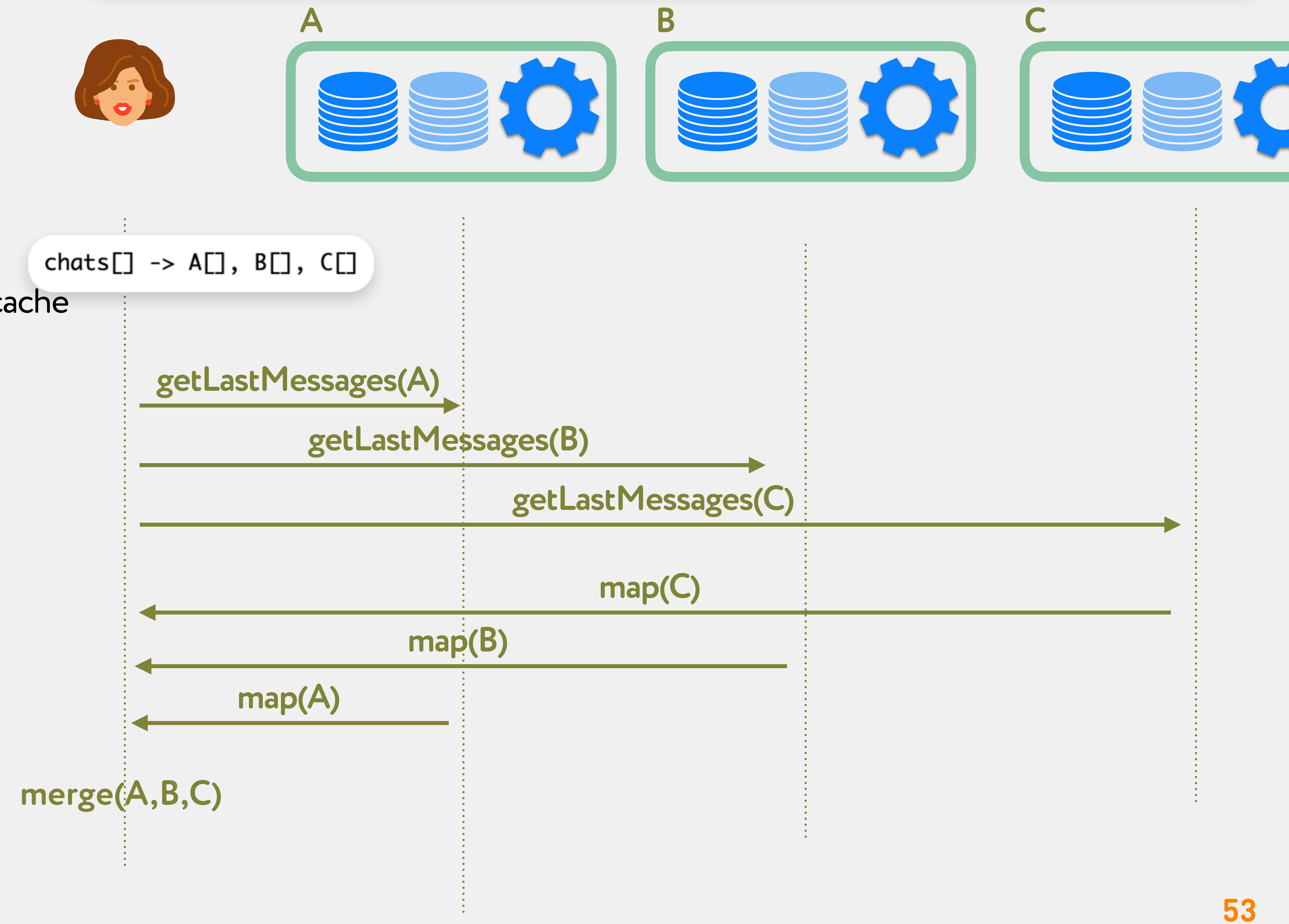
- **Multiple chats in request**
  No single node owns all data
- **Fraction of them are in cache**
  Meaningless to load inactive chats to cache



split & merge

```
Map<Long, Message> getLastMessages( long[] chatIds )
```

- **Multiple chats in request**
  No single node owns all data
- **Fraction of them are in cache**
  Meaningless to load inactive chats to cache



Full Text Search : search and indexMessages

- **Inverted index**

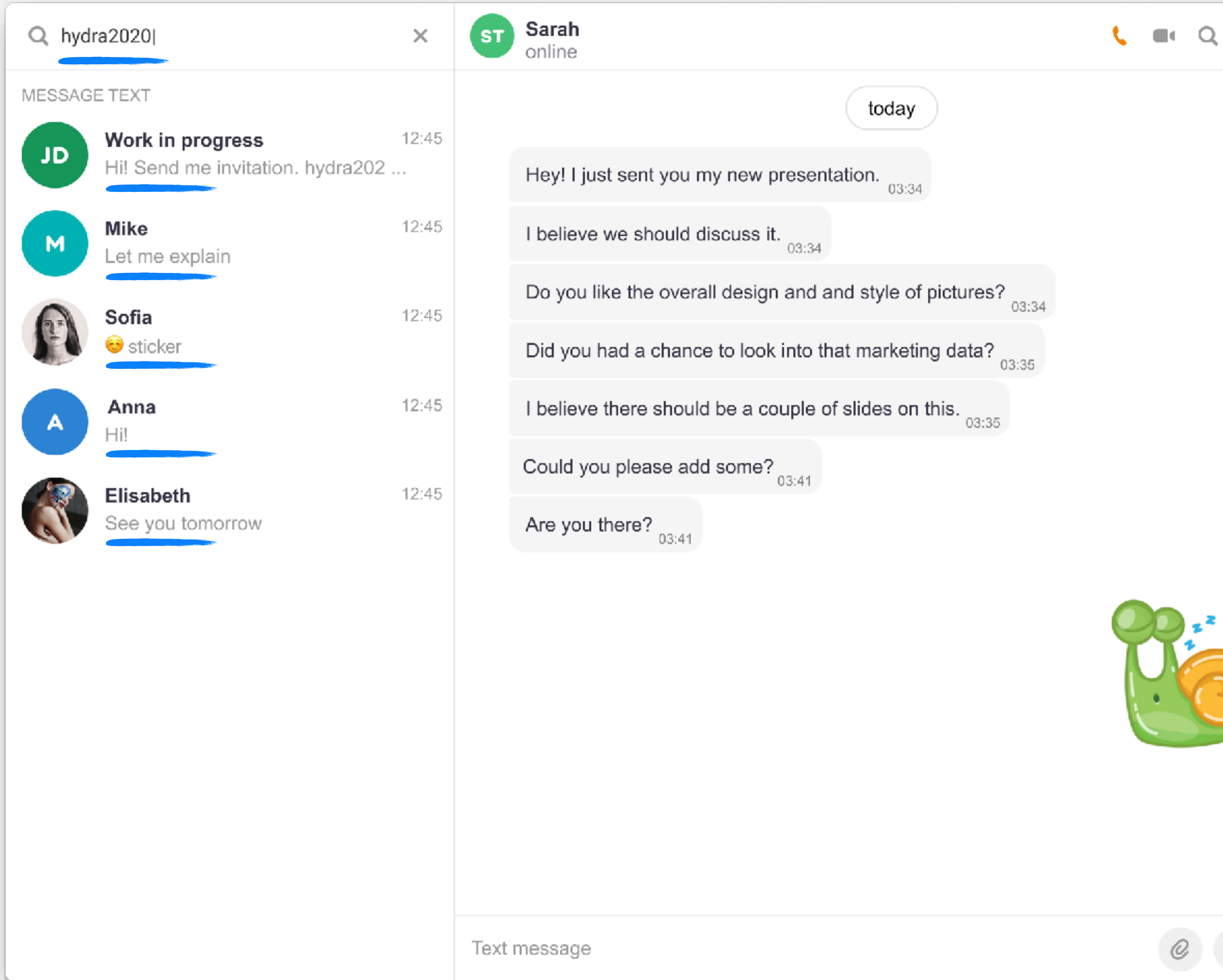
  lucene.apache.org

- **One per conversation**

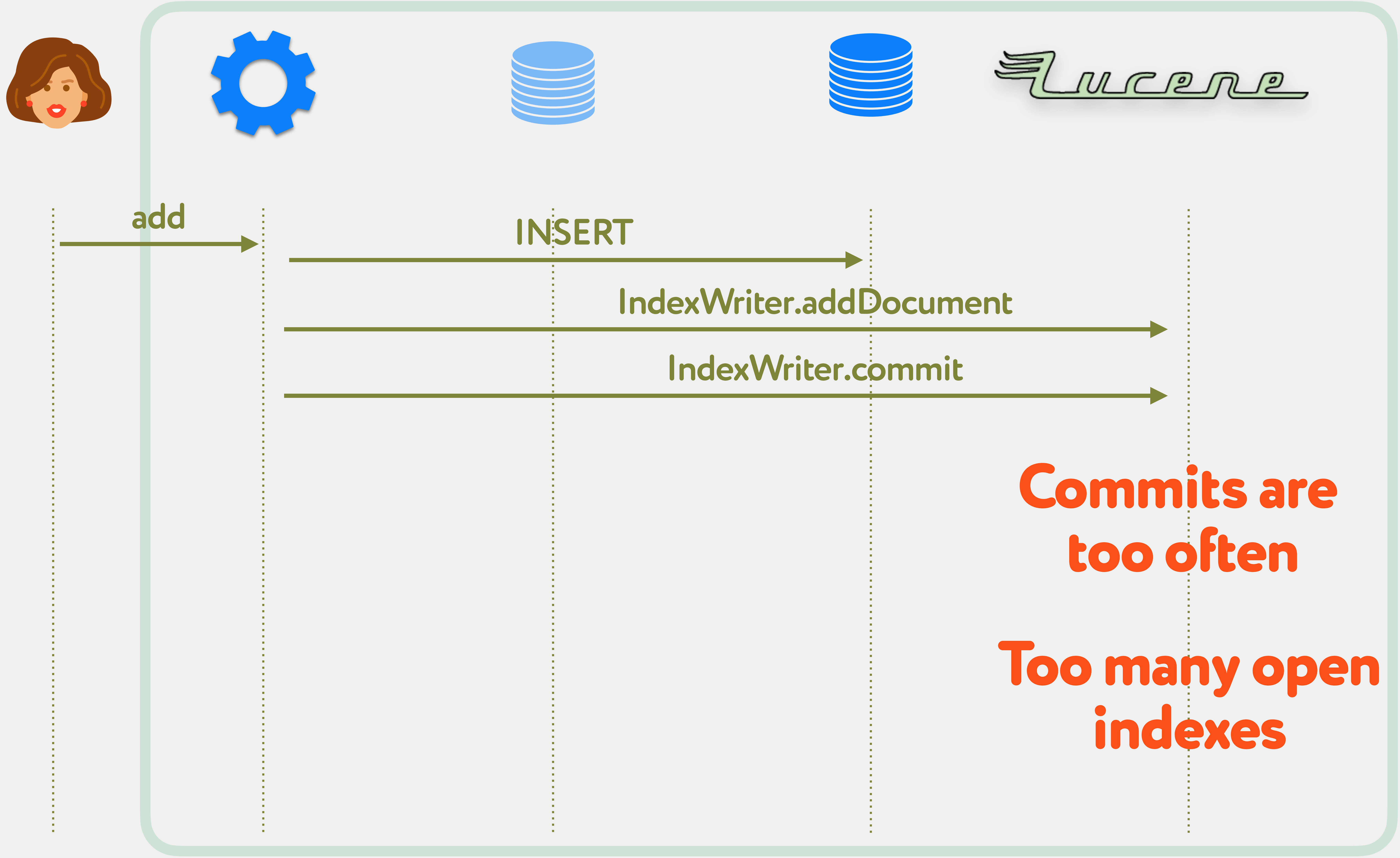
  single 100TB index does not work;
  per-user duplicates data

- **Large conversations only**

  Short chats index builds right before search



indexMessages



Compaction

- **Merges data generations**

  Across sstable files

- **In defined order**

  token(PartitioningKey), Clustering Key

```
package org.apache.cassandra.db.compaction;

public class CompactionManager implements CompactionManagerM
{
```

App to DB tight integration = even less costs

(8) http://cassandra.apache.org/doc/latest/operating/compaction.html

4 Operations

Longer deployments

what takes time:

- **DB initialization**

  Depends on the disk speed and the volume of WAL to play

- **In-memory lost state load**

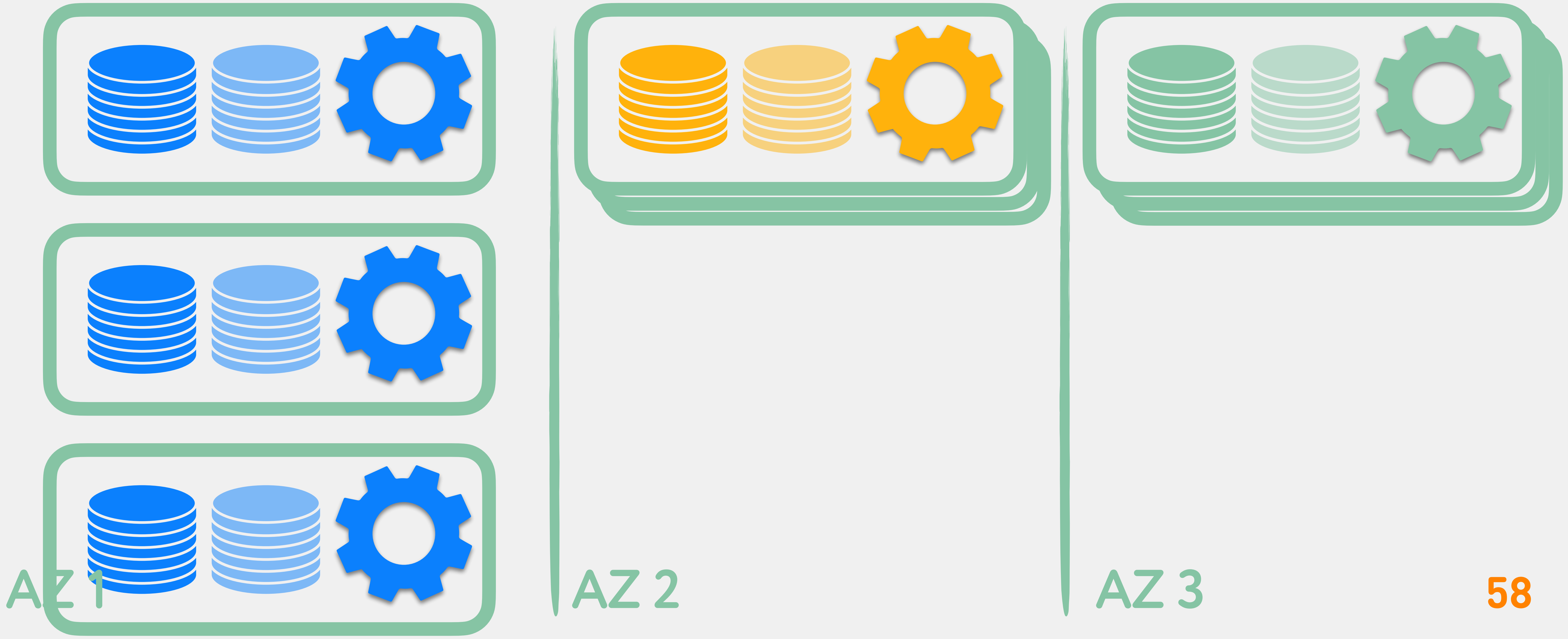
  Cache size, contention, CPU

- **Consistency wait time**

  Number of lost mutations

how to mitigate:

- **Parallel deployment of all availability zone instances**



More DB nodes restarts

why:

- **DB is embedded into app**

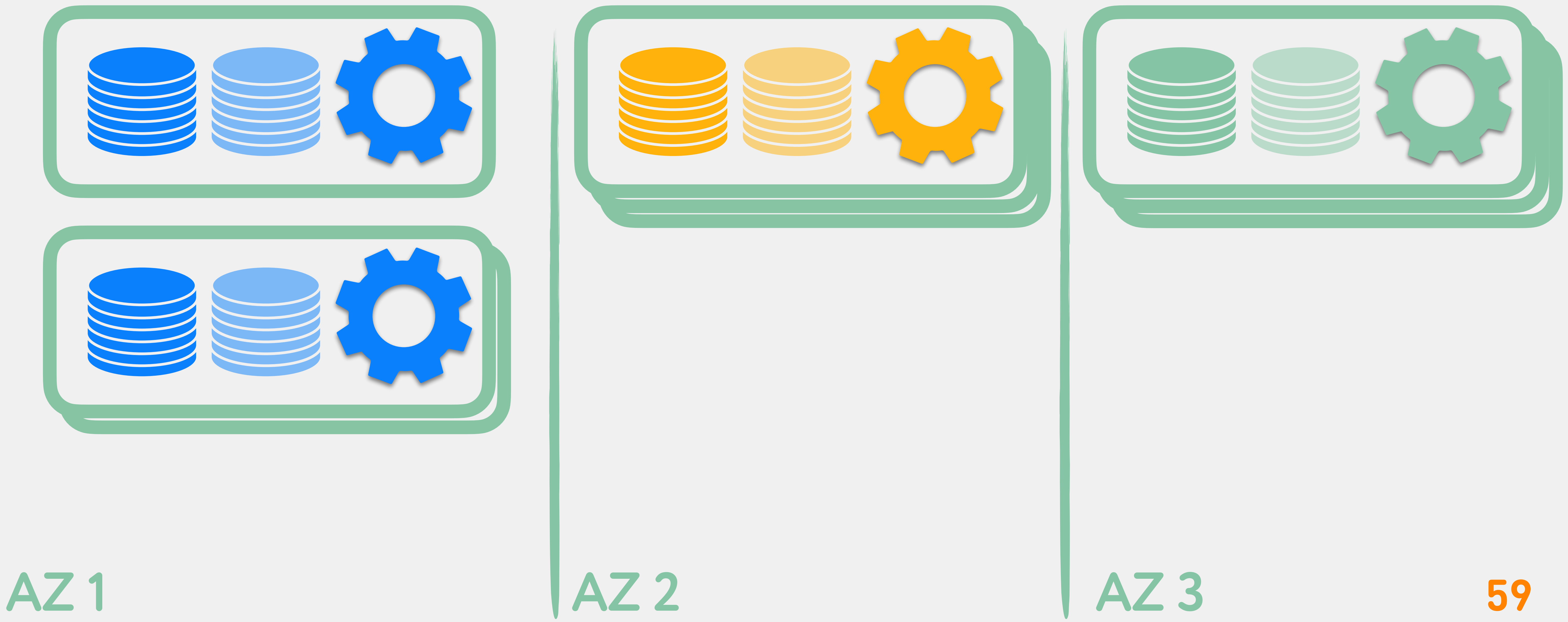
  app restart == DB node restart

not a problem

- **Nothing, this is good**

  Makes it easy to debug failures in controlled environment;

  Chaos monkey on a regular basis at no cost



Longer scaleout time

why:

- **state is colocated with application**

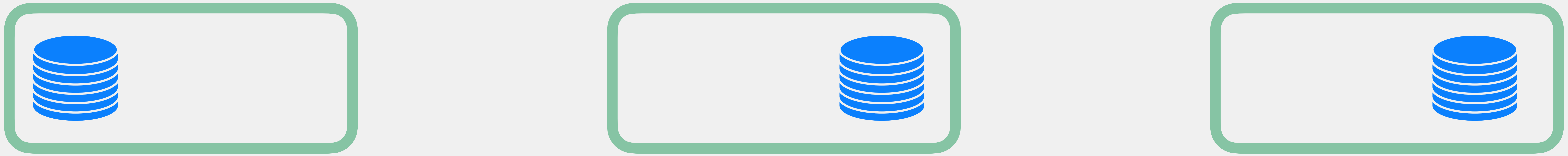
  Scaleout includes data resharding

how to mitigate:

- **Capacity planning**

  Scaleout upfront

- **Feature flags in right places**



Imbalanced resource utilization

why:

- **state is colocated with an application**

  Number of app, cache and db are equal

how to mitigate:

- **Container orchestration**

  one-cloud, k8s, aurora, mesos

One-cloud — the datacenter OS of ok.ru (Russian)
Oleg Anastasyev, 2018

Diagnostics and support

(9) https://github.com/jvm-profiling-tools/async-profiler

Stateful Services Summary

- **More effective and reliable**

  avoid marshalling, overreads and network costs

- **Caches are now consistent with data**

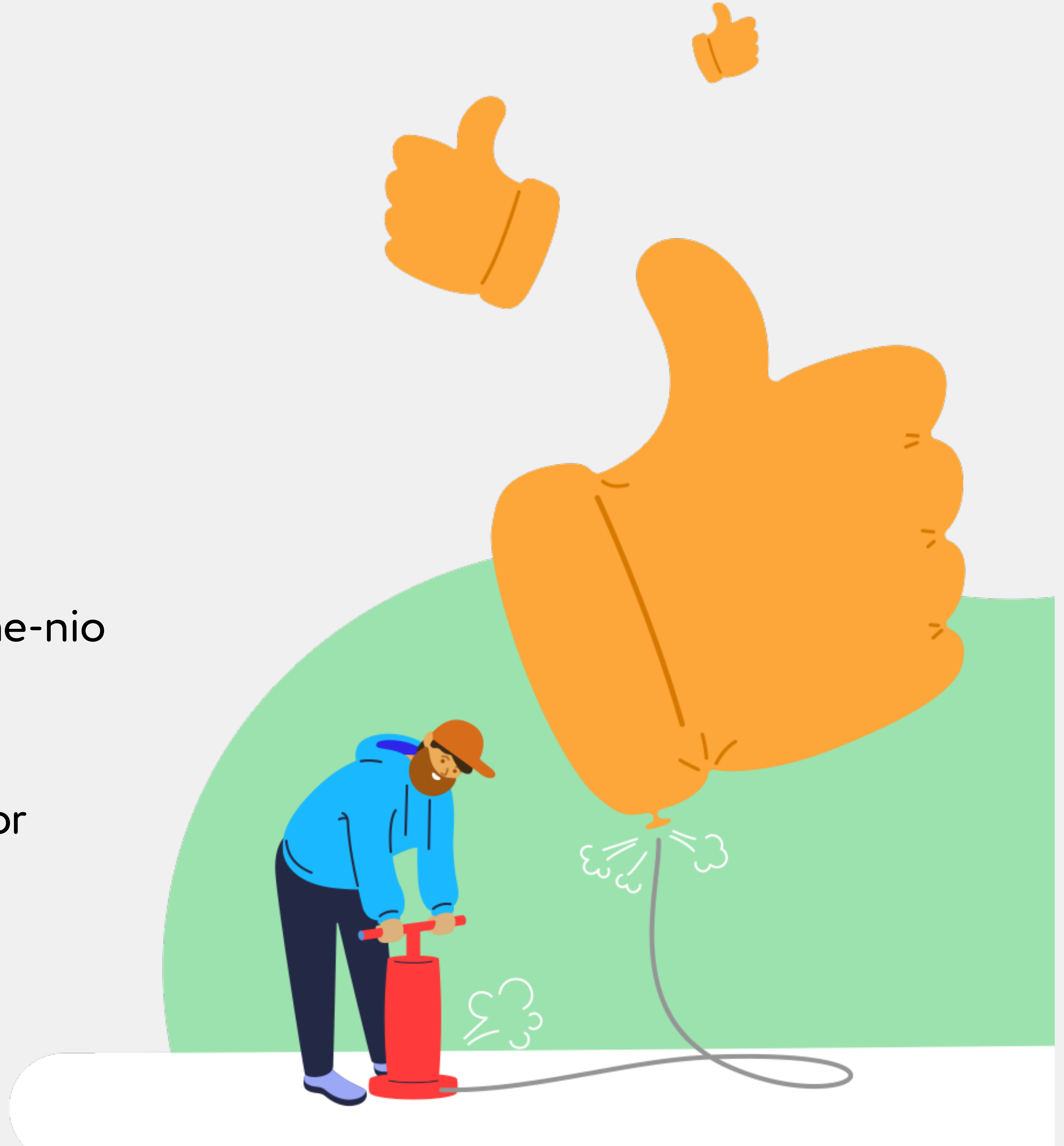
  Cache and C* are embedded in the single process

- **Not so hard to implement**

  Really hard parts are already implemented in C* and one-nio

- **Stateless is starting to obsolete**

  there are new solutions to the problems it was aimed for



Effective and Reliable Microservices

Oleg Anastasyev

oa@ok.ru

@m0nstermind